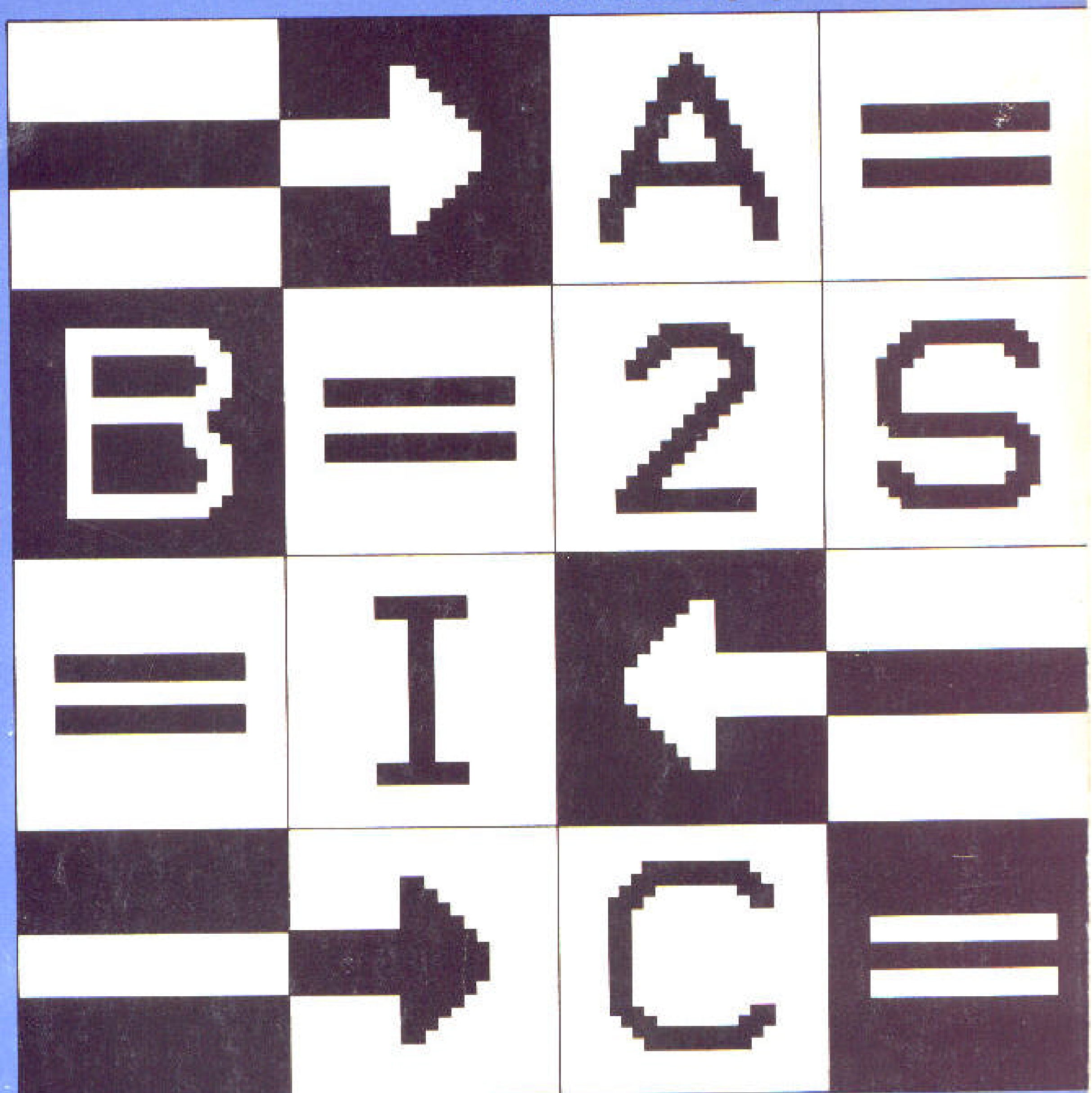
Texas Instruments TI-99/4A Computer

Beginner's BASIC

Step-by-step hands-on approach to learning the fun and power of programming in the TI BASIC language.

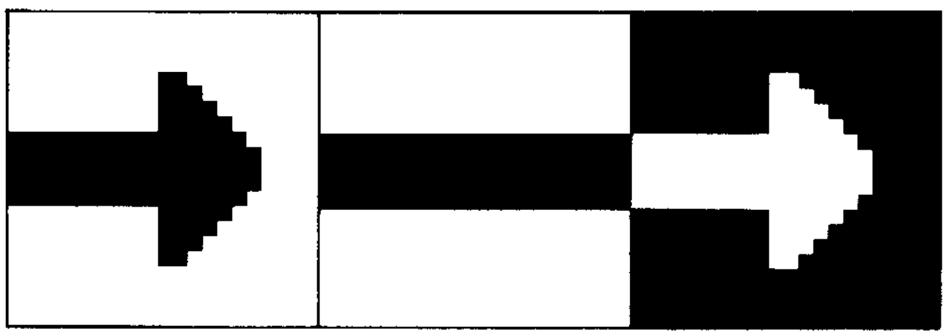


LCB 4180

Beginner's BASIC

A step-by-step guide that takes you from the "ground up" into an adventure – the adventure of communicating with a computer in a simple, yet powerful language.

Even if this is the first time you've seen a computer, you'll be able to follow this easy-to-understand, hands-on approach.



Note: The instructions and sample programs in this book are designed for use with the Texas Instruments TI-99/4A Computer. The information included will be generally useful with other computers incorporating BASIC programming language conforming to the American National Standard for Minimal BASIC. However, the program instructions included here – especially those for graphics and sound – will apply specifically to the TI computer.

This book was developed by

Don Inman
Ramon Zamora
Bob Albrecht

in cooperation with Texas Instruments Incorporated and the Staff of the Texas Instruments Learning Center:

Jacquelyn Quiram
Bob O'Dell

Artwork and layout were coordinated and executed by

Schenck Design Associates, Inc.

ISBN 0-89512-028-3
Library of Congress Catalog Number: 79-65510

Table of Contents

Introduction

You and Computer Programming

This book is your guide to an adventure — learning to program your Texas Instruments computer. Even if you have never worked with a computer before, you can use this book to teach yourself, your family, and your friends how to use and enjoy your computer.

Computers are rapidly becoming an everyday part of our lives. We're very familiar with bank statements compiled and printed by computers; we watch computerized scoreboards at sports events — we even see computer-aided instruction in our children's classrooms! Almost everything we eat, wear, and use has been handled at some point in the manufacturing process by computer-controlled machinery.

By learning to program and interact with your computer, you'll be joining this technological revolution of the Computer Age. You'll understand more about how computers work, what they can (and can't) do, and why they are becoming so widely used. Best of all, you'll be able to apply the power of your computer to the areas that appeal to you — your business and finance, your hobbies, your family's needs and interests.

And perhaps we'd better warn you — many people become fascinated with computer programming as an exciting and entertaining hobby! So don't be surprised when you — and all the family — find yourselves wanting to learn about and use your computer more and more as time goes on.

What is computer programming? Nothing mysterious! Programming is simply communicating with a computer — telling it what to do and when to do it. To program your computer you'll only need to learn two things: the language your computer understands, and the way you talk to it. No lengthy training periods or super-sophisticated skills are required.

TI BASIC

In order to communicate with any computer, you'll need to learn its language. The language we'll be exploring here is a form of BASIC (short for *B*eginners *A*ll-purpose *S*ymbolic *I*nstruction *C*ode). BASIC was developed by John Kemeny and Thomas Kurtz at Dartmouth College during the middle 1960's. Although BASIC is only one of many computer languages, it is one of the most popular in use today. It's easy to learn and simple to use; yet it's powerful enough to do almost anything you'd want to do with computers.

Some BASIC features may vary slightly from one type of computer to another. The similarities, however, far outweigh the differences; in fact, you can think of these different forms of BASIC as dialects of the same language. The dialect used by the TI computer is called TI BASIC.

As you read this book and try out the examples on your own computer, you'll notice one striking fact about BASIC: it's very much like English! You'll see words like PRINT, GO TO, RUN, and END. The meanings of these words in BASIC are almost identical to the definitions you already know and understand. This fact is what makes BASIC so easy to learn and fun to use.

Introduction

Now, how do you talk to the computer? Well, take a look at the keyboard. You see there the letters of the alphabet, numbers, punctuation marks, and other special symbols, many of which you've seen before on typewriter keyboards. Everything you'll need to use to communicate with your computer is right there on the keyboard. You "type" your instructions, and the computer "hears" them. It's essentially as easy as that!

About This Book

This book will guide you step by step through the process of learning TI BASIC. While the book is not a complete textbook on BASIC programming, the material included here will give you a good foundation for the continued development of your programming skills. (Once you are familiar and comfortable with BASIC, you'll be ready for the more advanced material found in the *User's Reference Guide.*) Throughout the book, each explanation of a statement or command is followed by one or more examples for you to try out. Also, you can — and should — experiment with other examples of your own, to help you become thoroughly acquainted with the capabilities of your computer. You'll find some special sections marked **EXPERIMENT!** throughout the book. These are for you to try out on your own.

In the first chapter we'll explore some of the BASIC statements that can be performed in what's called the *Immediate Mode* (that is, directly from the keyboard). Do you want to add 3 and 5, or create sounds, or make designs (computer graphics) on the screen? You can do all of these in the Immediate Mode, and you'll find out how to do them in Chapter 1.

Chapters 2 and 3 take you on into programming. You'll learn how to "structure" a program, issue "commands" your computer can follow, perform mathematics, use graphics and sound more effectively, and create loops (program segments that repeat themselves).

Then, in Chapters 4 and 5, you'll get further into some of the exciting things a computer can do. Did you know that your computer can play games? Make music? Draw colorful designs on the television screen? It can, and you can teach it how!

At the end of the book are several appendices of reference information you can use as you develop your own programs. Of special interest to those who want and need to use the computer as a powerful computational device, *Appendix D* outlines the mathematical operations and functions of the computer. You'll also find a convenient alphabetized index of topics to help you look up features you want to review.

Now that you know what's ahead, let's waste no more words — let's get started in the Immediate Mode.

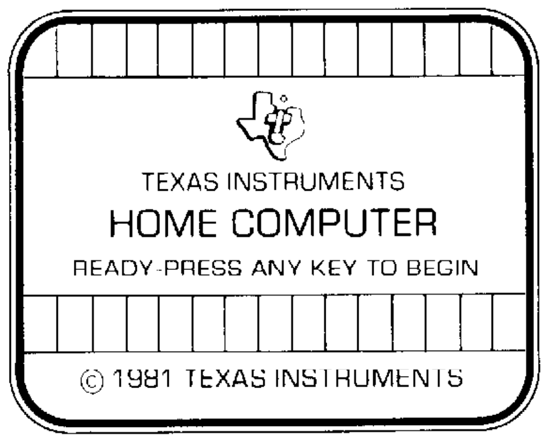
1 CHAPTER ONE

The Immediate Mode

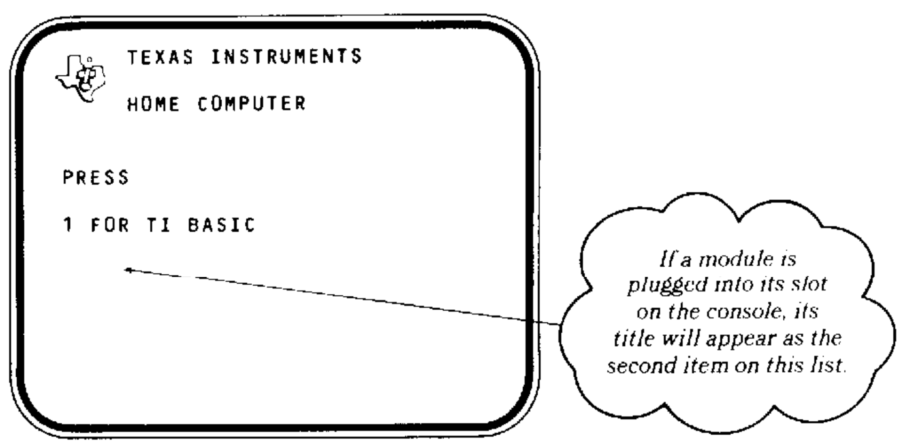
In its Immediate Mode, your computer "immediately" performs each BASIC statement you've typed in, as soon as you press **ENTER**. Because you can see an instant response on the screen, the Immediate Mode is a good way to introduce and explore certain BASIC language statements.

Before you begin learning BASIC, take a few minutes to review the operation of the keyboard. You'll find a complete "key tour" in the *User's Reference Guide*.

When you are ready, turn on your computer. The display screen should look like this:



Press any key on the keyboard. The display will then show the master selection list.



(*Note:* When you're ready to leave TI BASIC, just type the word **BYE** and press the **ENTER** key. The computer will then return to the main title screen.)

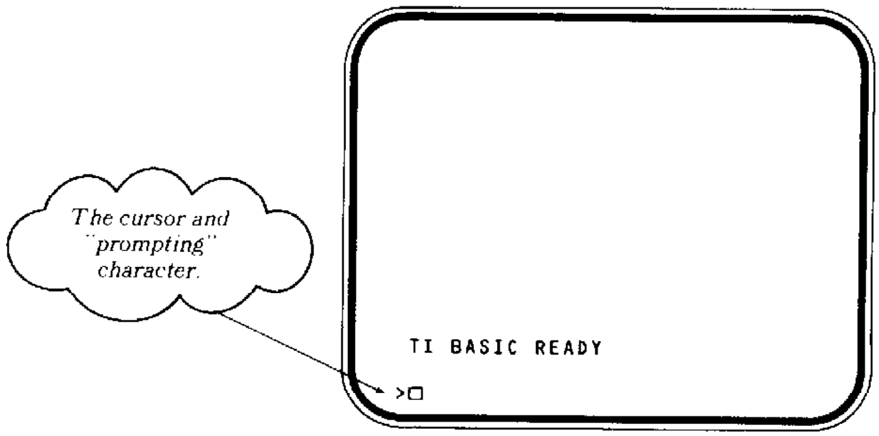
The examples shown in this book are printed in upper-case (large capital) letters. If you want to reproduce the examples exactly as you see them here, press down the **ALPHA LOCK** key. However, in most cases the computer accepts either upper-case or lower-case letters.

Also, see *Important Keyboard Information* on the inside front cover of this book for details about the function keys (**CLEAR**, *left-arrow* key, *right-arrow* key, etc.).

1

Press the **1** key to select TI BASIC.

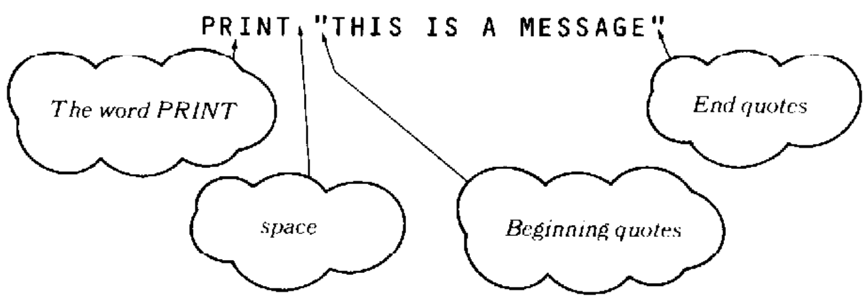
The display now shows that the computer is ready for you to begin.



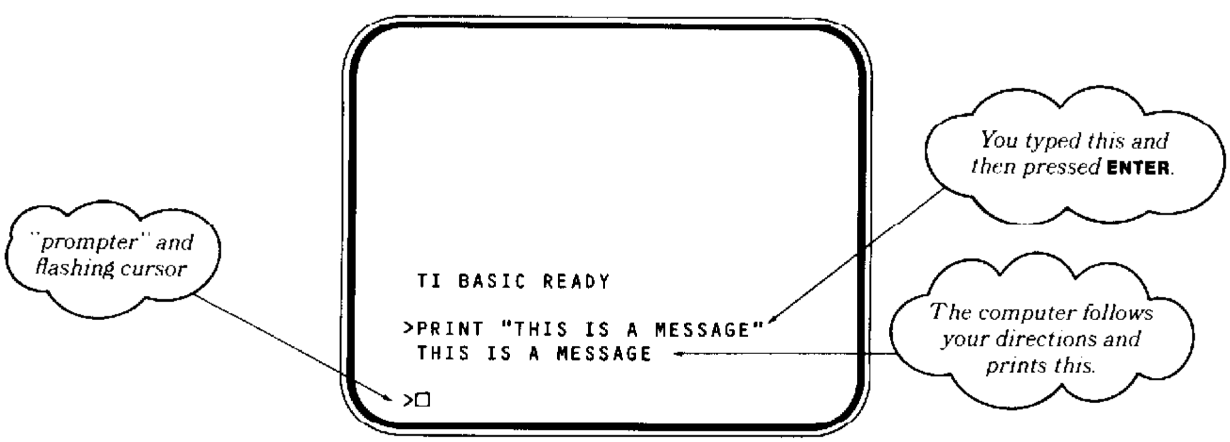
The flashing rectangle is called the *cursor.* It tells you that the computer is ready for you to use. Whenever you see the cursor, you know that it's your turn to do something. The prompting symbol marks the beginning of each line you type.

The PRINT Statement

The PRINT statement means exactly what it says. You merely type the word PRINT, followed by a message enclosed in quotation marks, and the computer prints the message when you press **ENTER**.

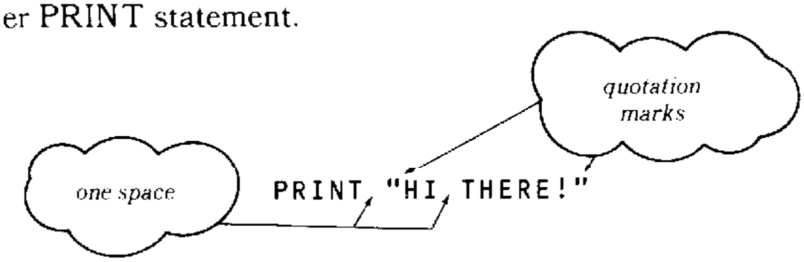


Remember to press the **ENTER** key after the ending quotation marks! This is the computer's cue to perform what you have requested.



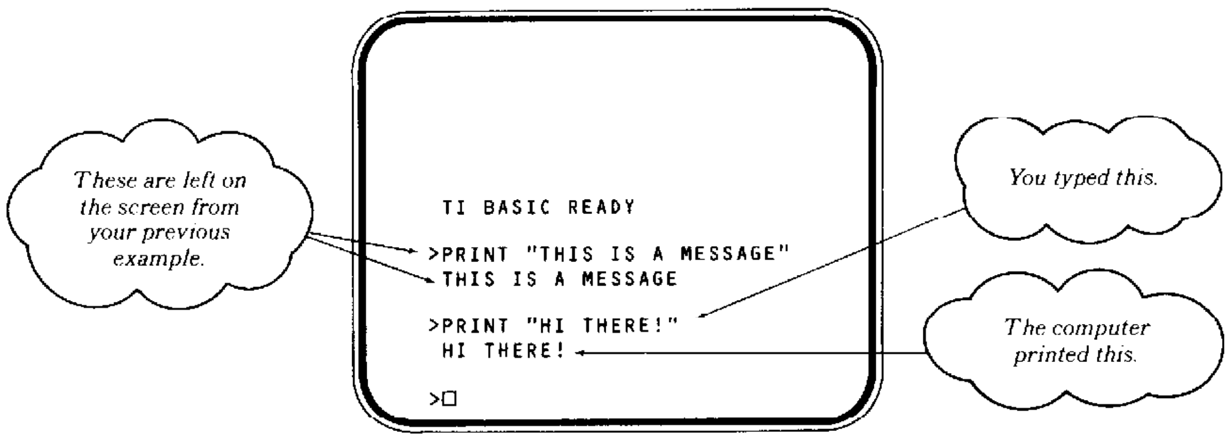
Let's try another PRINT statement.

Type this:



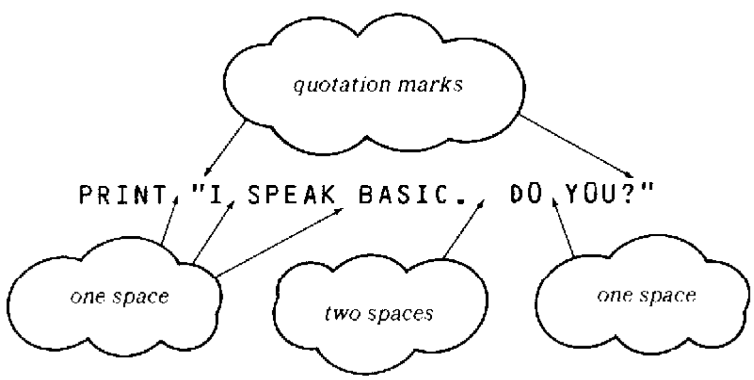
(*Note:* If you accidentally press a wrong letter or symbol key, just use the *left arrow* key to move the cursor back to the incorrect symbol. Then retype.)

Now press **ENTER**, and the computer will do just what you told it to do:



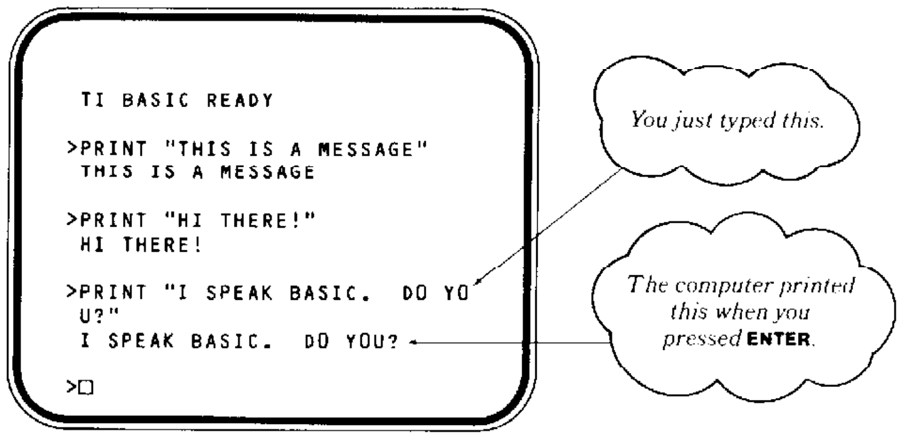
Did you notice the way the lines moved up on the screen when you pressed **ENTER** and again when the computer finished printing its line? This procedure is called *scrolling*. The cursor tells you it's your turn now and shows you where the next line will begin.

Let's try another example. Type these words, but don't press **ENTER** just yet:



(When you run out of room on a line, just keep typing – the computer will automatically "scroll" to the next line.)

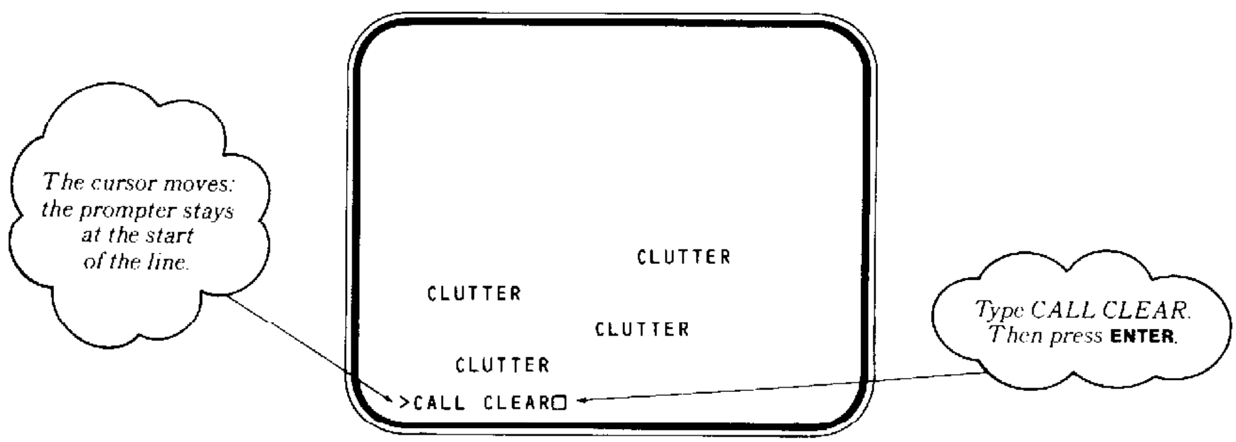
Now, look at the screen and check what you've typed. If there are any errors, just use the *left arrow* key until the cursor has reached the error. Then retype the line correctly from that point on. (This is only one correction procedure – you'll learn others as you go along in the book.) When everything is correct, press the **ENTER** key. You'll then see:



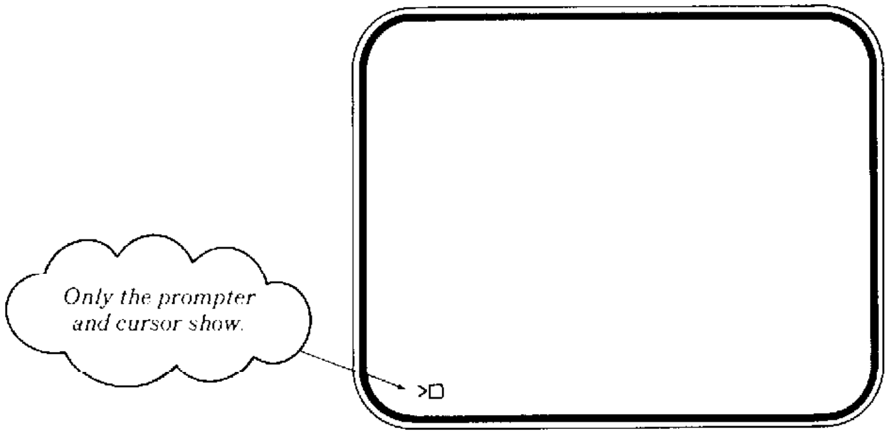
If you want to try some other PRINT statements on your own, go right ahead. Each time you press **ENTER**, you'll see the lines on the screen scroll upwards. The top lines will finally begin to disappear as the screen's capacity (24 lines) is reached.

The CALL CLEAR Statement

You've probably noticed that your video display has begun to look rather cluttered. If you want to clear the screen for a less distracting appearance, you can use the words CALL CLEAR.



CALL CLEAR wipes the slate clean for your next request, and your display will look like this:



Note: As you work through this book, you'll see several BASIC statements that begin with the word CALL. Your computer has been "taught" to do certain things by having some special-purpose programs built into it, and a CALL statement tells the computer to "call" the built-in program named in the statement.

Error Messages

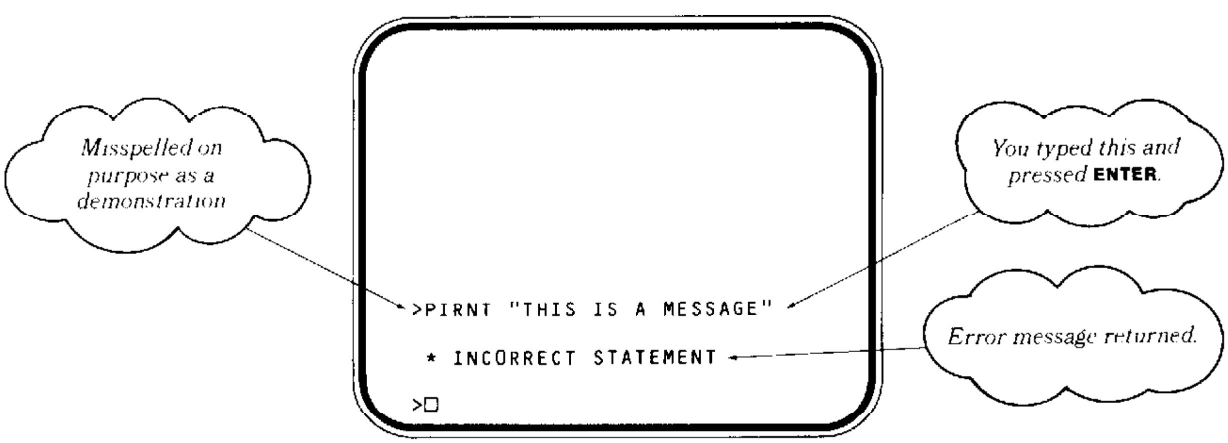
Every computer programmer makes mistakes, so don't hesitate to try experiments of your own as you go through the examples in this book. Errors will not hurt the computer. It quickly recognizes things it can't do and gives you an error message and a tone to tell you to try again. When mistakes happen, just identify the error and retype the instruction correctly.

Some of the most common errors are typing a wrong letter and omitting a necessary part of the statement. For example, here are a few mistakes your computer doesn't like in a PRINT statement:

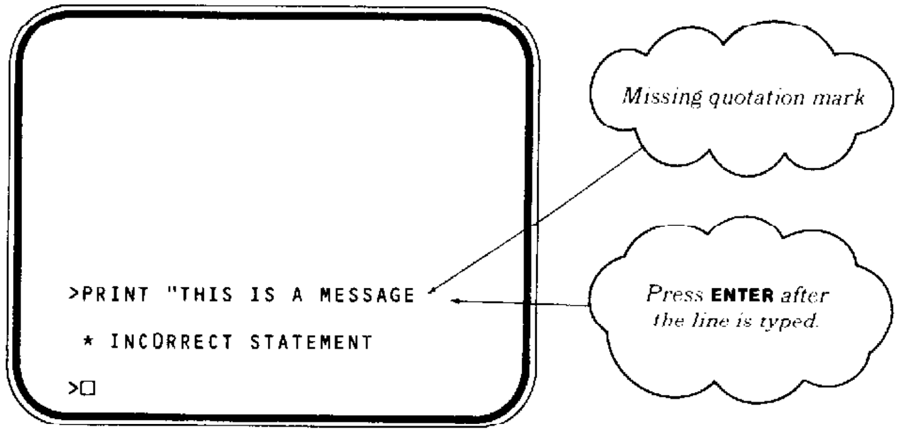
1. A misspelling in the word PRINT.
2. A missing or extra quotation mark.
3. Extra spaces in the word PRINT.

Let's experiment with some intentional errors to become more comfortable with error messages.

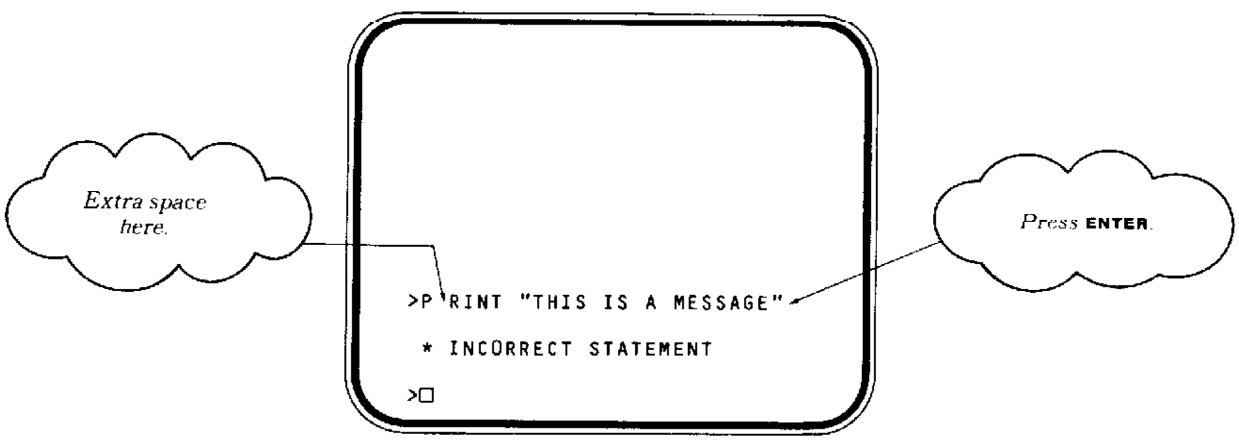
(1) Misspelling in the word PRINT



(2) Missing or extra quotation marks



(3) Extra spaces in the word PRINT



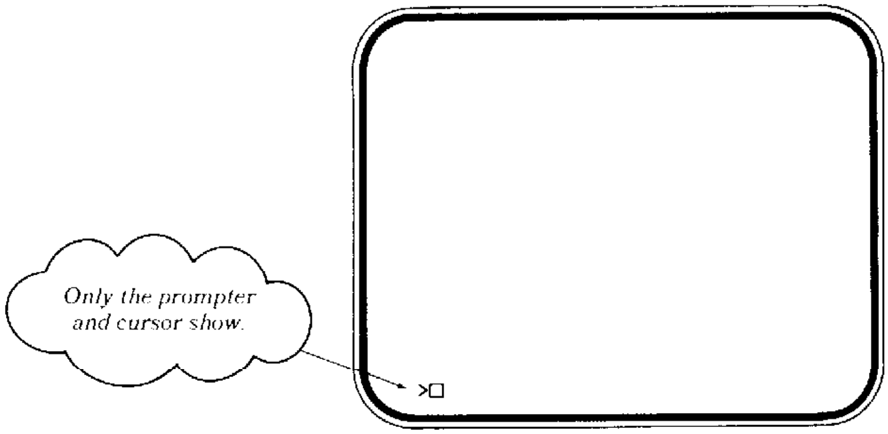
Experiment!

Try a few more messages with the PRINT statement, introducing intentional errors so that you will become familiar with the error messages. (We'll discuss other error messages at appropriate places throughout the book.).

Error Correction

There are several ways to correct typographical errors *before* you have pressed **ENTER**.

1. You can press **ERASE** to erase what you've typed on the line.
2. If you spot the mistake just after you've made it, use the *left arrow* key to move the cursor back to the error, retype the line from that point on, and then press **ENTER.** (Note that the characters are not erased as you backspace over them.)



Note: As you work through this book, you'll see several BASIC statements that begin with the word CALL. Your computer has been "taught" to do certain things by having some special-purpose programs built into it, and a CALL statement tells the computer to "call" the built in program named in the statement.

Error Messages

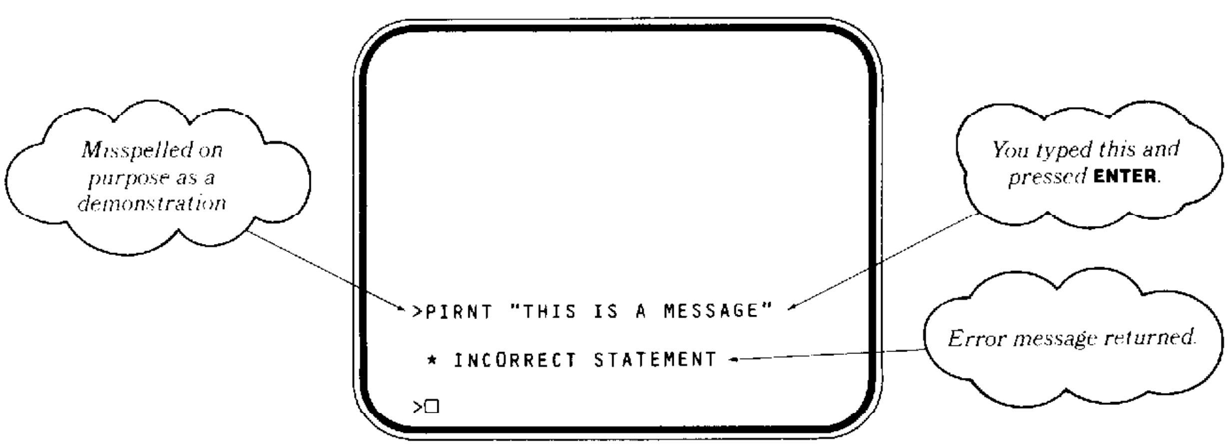
Every computer programmer makes mistakes, so don't hesitate to try experiments of your own as you go through the examples in this book. Errors will not hurt the computer. It quickly recognizes things it can't do and gives you an error message and a tone to tell you to try again. When mistakes happen, just identify the error and retype the instruction correctly.

Some of the most common errors are typing a wrong letter and omitting a necessary part of the statement. For example, here are a few mistakes your computer doesn't like in a PRINT statement:

1. A misspelling in the word PRINT.
2. A missing or extra quotation mark.
3. Extra spaces in the word PRINT.

Let's experiment with some intentional errors to become more comfortable with error messages.

(1) Misspelling in the word PRINT



3. If you've finished typing a line and you find a mistake near the beginning of the line, use the *left arrow* key as above, retype the letter or word, use the *right arrow* key to move the cursor back to the end of the line, and then press **ENTER**. Note that the *right arrow* key *does not* erase as it moves the cursor. If you need to erase a character or word, use the **SPACE BAR** to advance the cursor over the character.

OR

You can just disregard the error and press **ENTER** anyway. The computer may give you an error message, but it's very forgiving. Simply retype your line – correctly, this time – and press **ENTER** again.

The LET Statement

The LET statement is used to assign a value to a *variable*. Variables are "names" given to numbers or to phrases containing both numbers and letters (and certain other characters). Although there are two types of variables, in this section we'll consider only those variables that give names to numbers. These are called numeric variables. A numeric variable is just a name given to a numeric value.

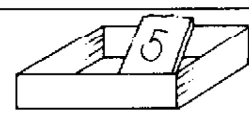
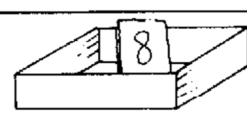
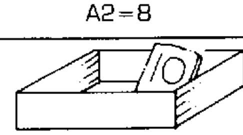
In the LET statement the word LET is followed by the variable (the name), then an equals sign, and finally the numeric value you're assigning to the variable. Variables can be up to 15 characters long, but they are generally kept fairly short for convenience.

Let's try a few examples. Type in the following lines, pressing **ENTER** at the end of each line:

one space

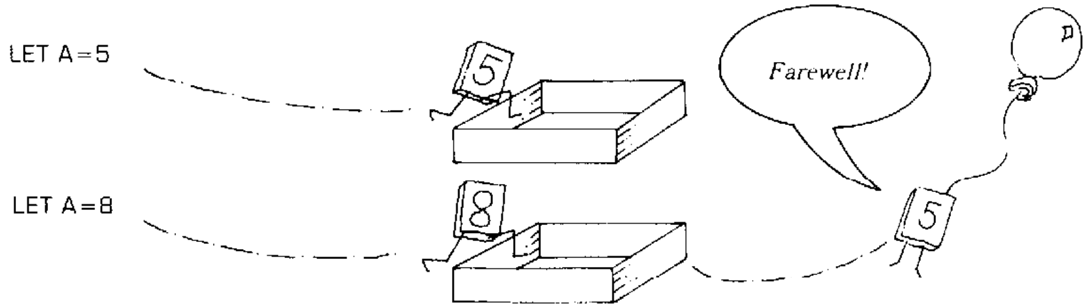
```
LET A=5
LET A2=8
LET ALPHA=10
```

You can think of variables as labeled boxes that hold assigned values.

LET A-5	 A=5
LET A2=8	 A2=8
LET ALPHA=10	 ALPHA=10

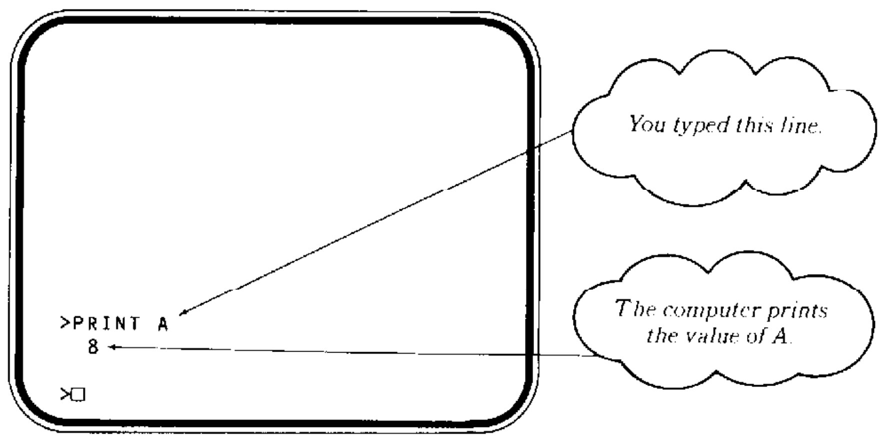
1

Only one value at a time may be assigned to a given variable, but you can change a value easily. Type these successive LET statements, pressing **ENTER** after each line.



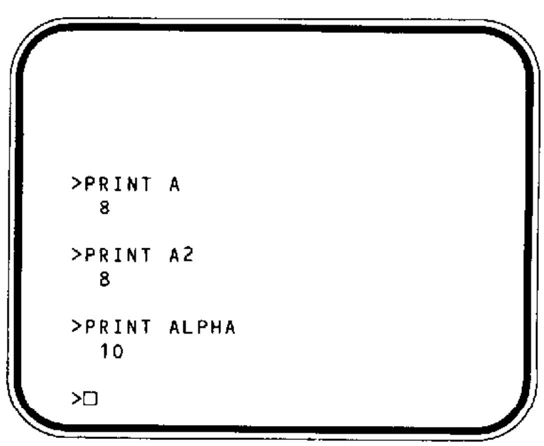
The value of A is no longer 5. The 5 has been replaced by the value 8.

Now let's use PRINT statements to check the values we've entered. Clear the screen; then type PRINT A and press **ENTER**.

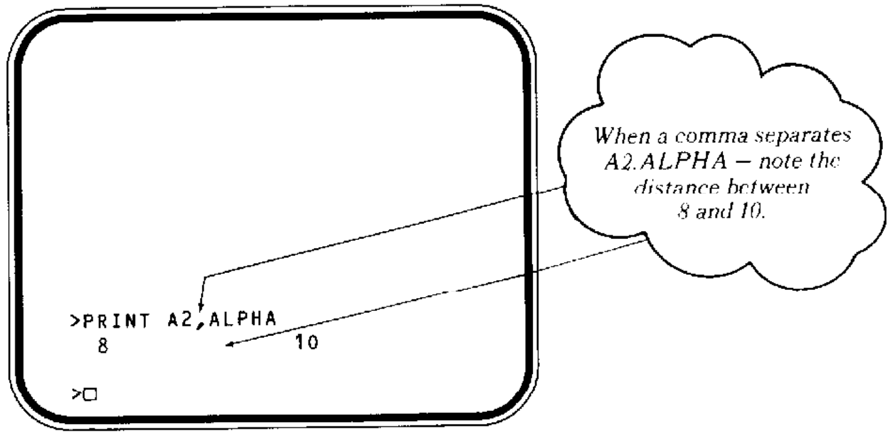


Did you notice that this PRINT statement is different from the PRINT statements we explored earlier? We didn't put quotation marks around the A. That's because we didn't want to print the letter A; we only wanted to see the numeric value assigned to A.

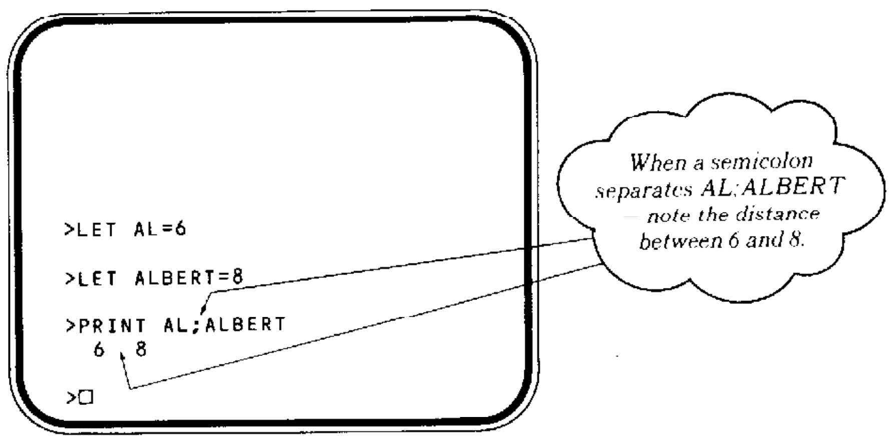
Now, check for the values of A2 and Alpha. (Remember! Press the **ENTER** key at the end of each line, even though it isn't shown in the illustration below.)



A single PRINT statement can also be used to print two or more things. Clear the screen, and try these examples:

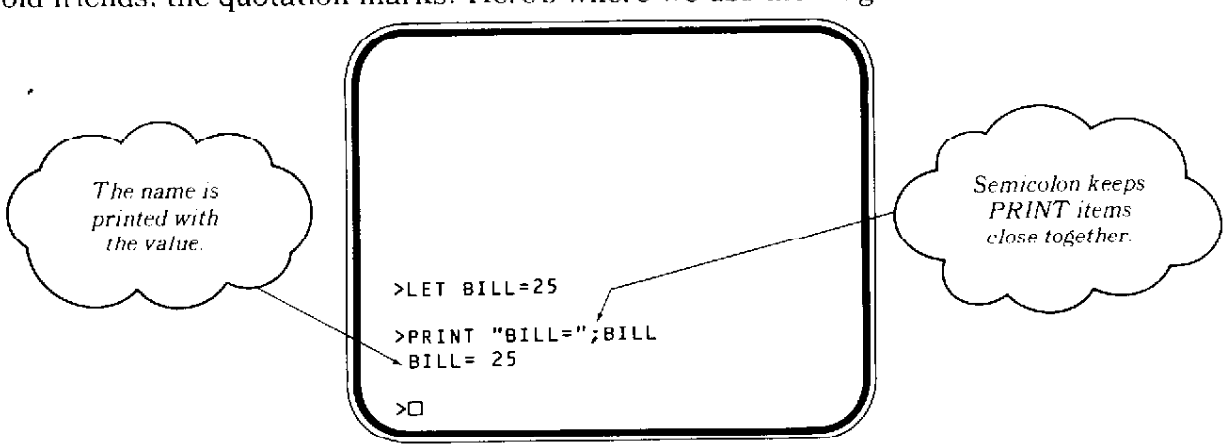


Now, try these:



The computer divides the display screen into two horizontal zones. When you use a comma (,) between two (or more) variables in a print statement, you are telling the computer to print the values in different zones. On the other hand, the semicolon (;) instructs the computer to print the numbers close together.

If you want to print the variable's name along with its value, you can. Remember our old friends, the quotation marks? Here's where we use them again:



(Did you remember to press **ENTER** at the end of each line?)

1

Now that you've learned to assign values to variables, what can you do with this new skill? Let's find out. First, use the CALL CLEAR statement to clear the screen.

After variables have been assigned values by LET statements, the PRINT statement may be used to perform arithmetic operations on the variables and to display the results.

```
>LET W=4

>LET T-8

>PRINT W+T;T-W
  12  4

>□
```

You can also perform multiplication and division by using an asterisk (*) to multiply and a slash mark (/) to divide. For example,

```
>PRINT W*T;T/W
  32  2

>□
```

Note: In TI BASIC, the LET statement is not the only way to assign a numeric value to a variable. Your computer will also accept the assignment without the word LET:

```
>JACK=3

>JILL=5

>PRINT JACK*JILL
  15

>□
```

In other words, the word LET is optional in TI BASIC; your computer will accept the assignment either way.

Experiment!

Try other variable names and numeric values, and experiment with using the comma and semicolon to separate variables in the PRINT statement. Try adding, subtracting, multiplying, and dividing these variables in PRINT statements. Discover what mistakes will cause error messages.

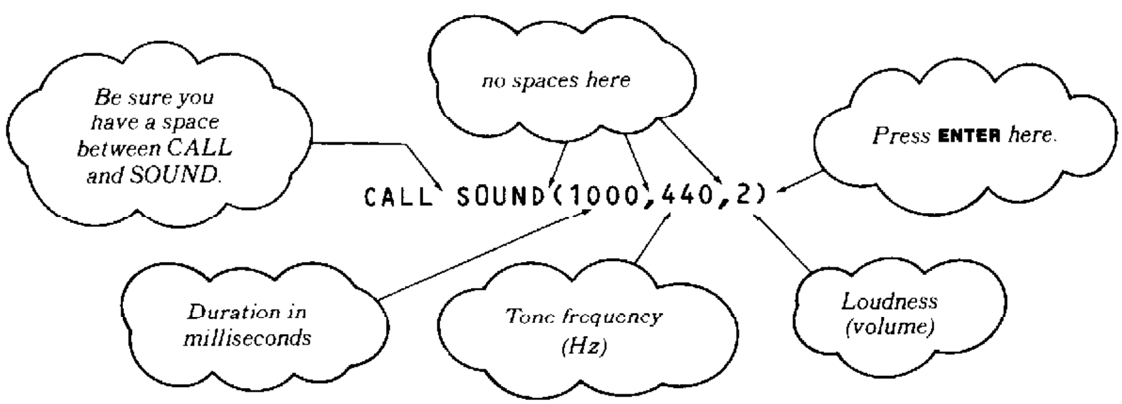
The CALL SOUND Statement

Here is another of the CALL statements. (Remember the CALL CLEAR statement we discussed earlier? We hope you've been using it occasionally to "erase" the display screen.)

Using the CALL SOUND statement, you can produce sounds over a range of frequencies from 110 to more than 44,000 Hertz. One Hertz (abbreviated Hz) is equal to one cycle per second. Thus the sounds you generate with your computer can vary from 110 cycles per second (A below low C on a piano keyboard) to over 44,000 (well above human hearing limits).

You can also control the *duration* and the *volume* of the sound. The time the sound lasts (duration) ranges from 1 to 4250 milliseconds. One thousand (1000) milliseconds equal one second, so the duration range could be stated as being from 0.001 to 4.250 seconds. Volume selections are scaled from 0 to 30. Zero and one produce the same sound level and are the loudest. Thirty produces the quietest tone.

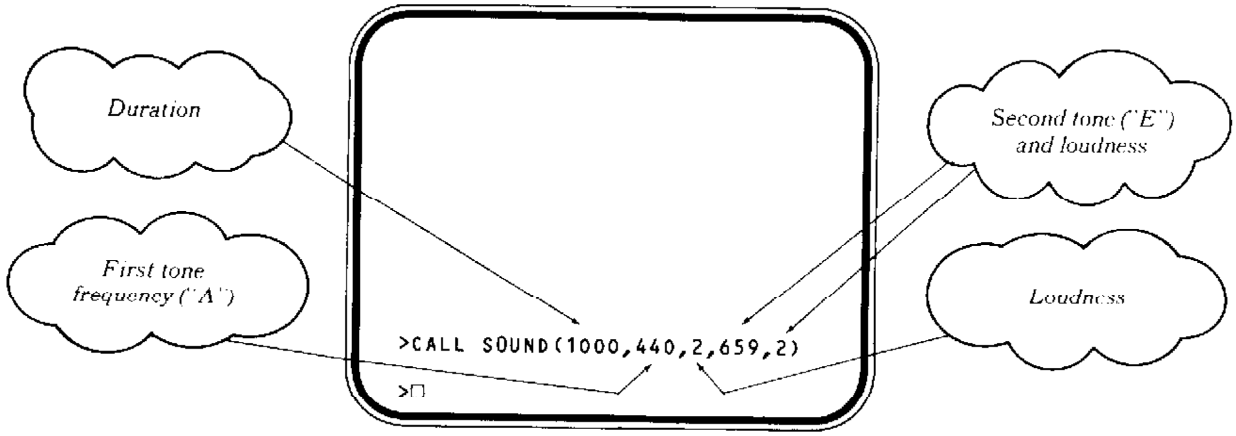
This example shows how to use the CALL SOUND statement:



Notice that the three values that control the sound are enclosed in parentheses following the words CALL SOUND. This example will produce a note of 440 Hz (A above middle C) with a duration of 1000 milliseconds (one second) and a volume of 2 (quite loud!).

Try the example now to hear the tone quality of your computer.

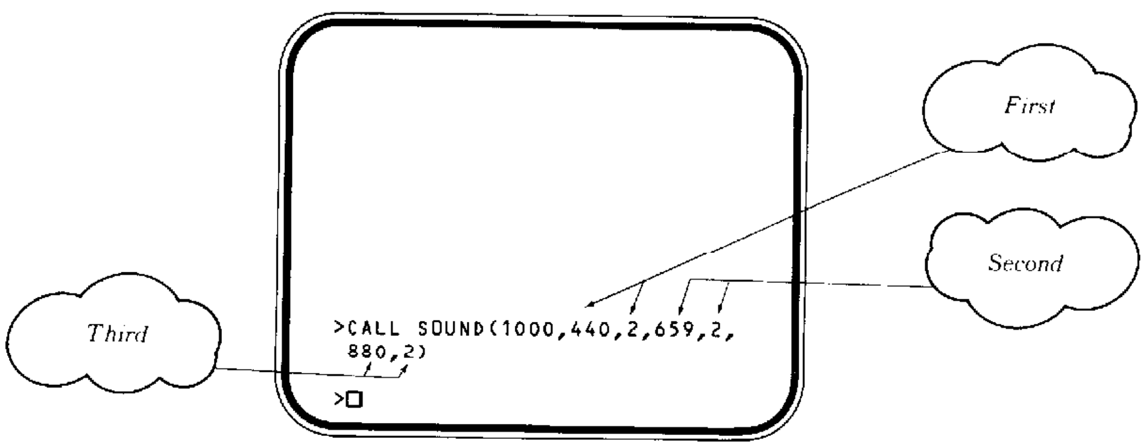
You can play more than one tone in a single CALL SOUND statement. Let's add a second note and see how this enhances the sound.



Note: Because the statement above contains exactly 28 characters (letters, spaces, and symbols), the cursor will move down to the next line as soon as you type the close parenthesis symbol. Be sure that you remember to press **ENTER**! (Notice that the prompting symbol stays at the beginning of your line.)

You only had to type the duration value (the number code that determines how long the sounds last) one time – at the beginning of the CALL SOUND instruction enclosed in parentheses. Both of the sounds *must* last for the same length of time. On the other hand, you *can* vary the loudness values. What would happen if you typed 5, instead of 2, for the second note's loudness? Try it!

Next, try a three-note chord:

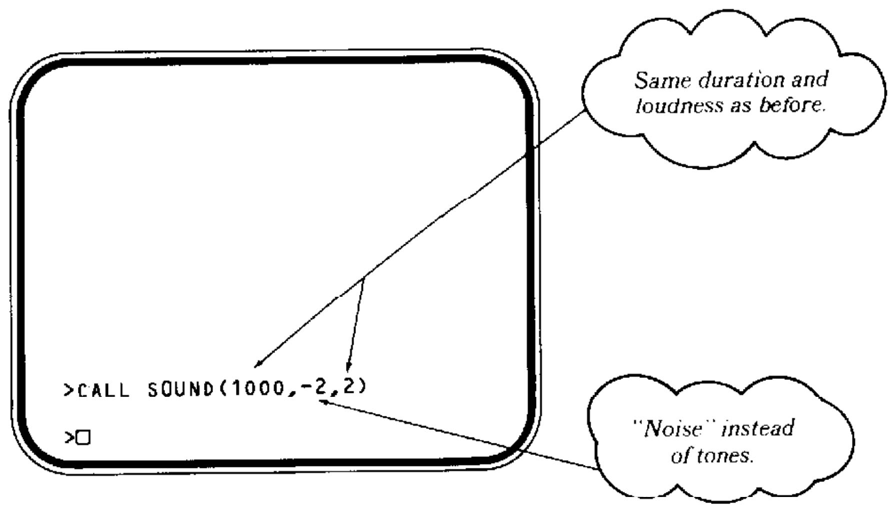


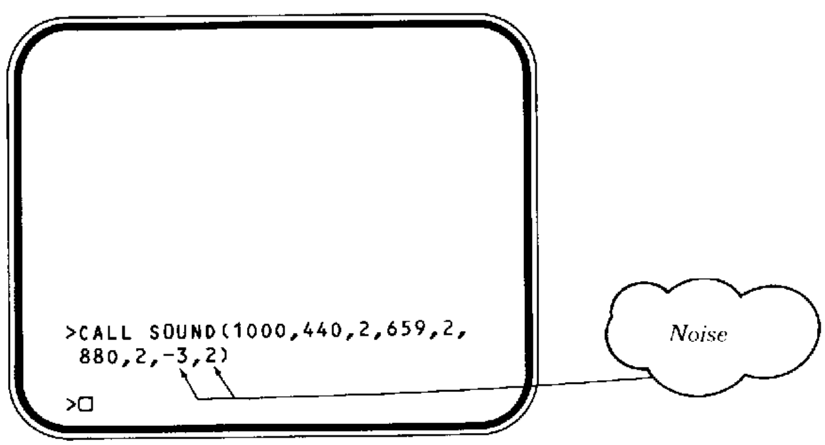
(Part of this CALL SOUND statement extends to the second line, since TI BASIC uses only 28 printing positions per line. This gives large, clear, readable text on the screen.)

You can produce up to three tones and one "noise" simultaneously over a given time duration. Noise is rather hard to define in words; it's best for you to experiment and hear for yourself. Remember, one person's "noise" may be another person's "music"!

To produce noise instead of tones, replace the tone frequency with a negative integer from -1 to -8.

Try these examples:





You can also use variables, rather than actual values, in the CALL SOUND statement. For example, let's use these variables:

T = time (duration)
V = volume (loudness)
C = 262 (Middle C on the piano)
E = 330 (E)
G = 392 (G)

So type in the following LET statements:

```
LET T=1000
LET V=1
LET C=262
LET E=330
LET G=392
```

Now you're ready for the CALL SOUND statement. Type:

```
CALL SOUND(T,C,V,E,V,G,V)
```

and press **ENTER**.

Experiment!

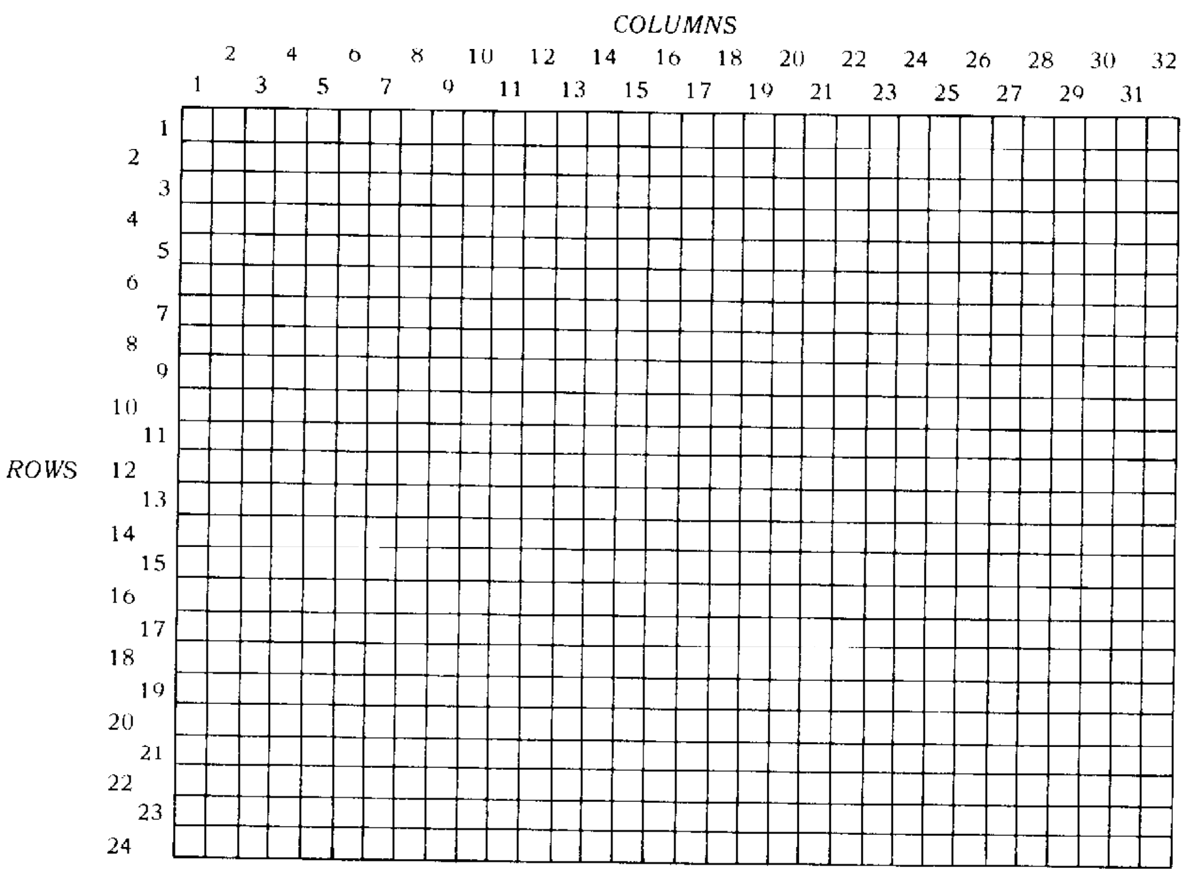
Experiment with other values for duration, tone, volume, and noise within the required range of values for each. (A list of musical note frequencies is included in *Appendix A*.) You'll soon be able to create imaginative sound effects for use in your future programs. The Immediate Mode is quite helpful for this type of experimentation.

Graphics (CALL VCHAR and CALL HCHAR)

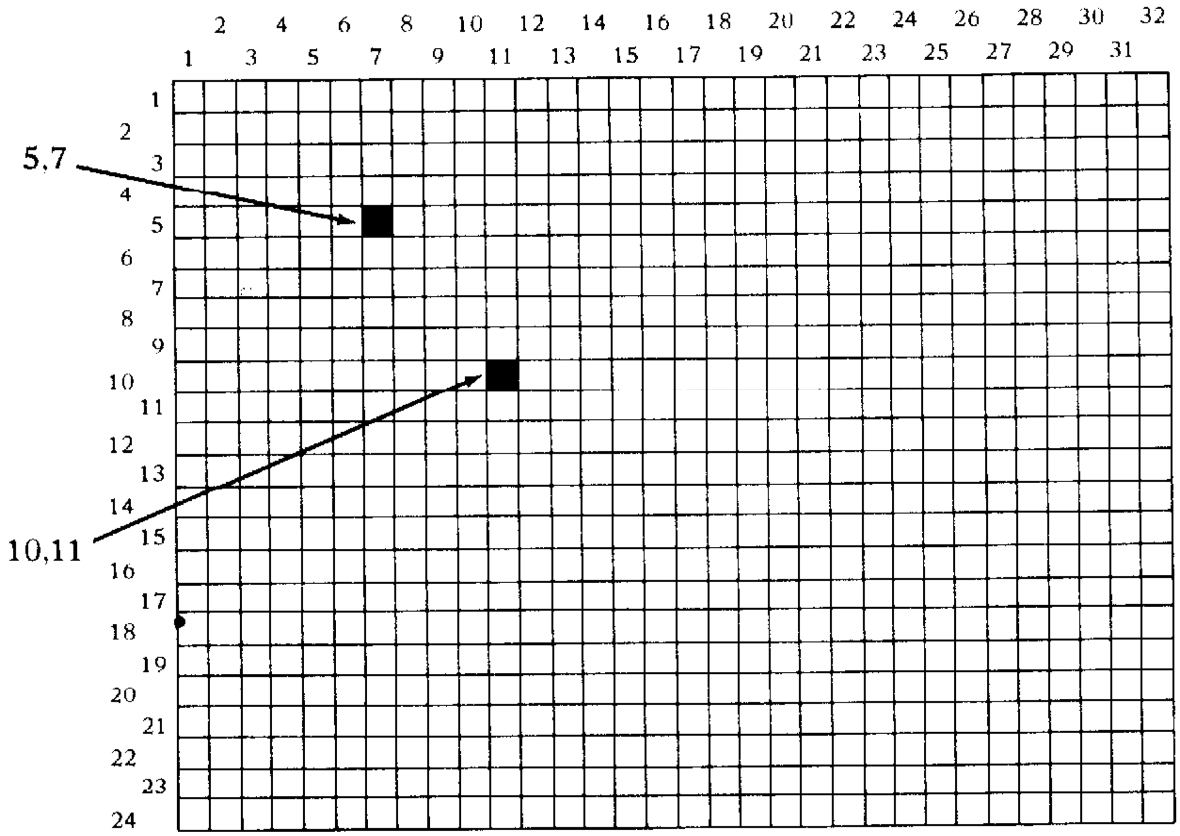
One of the most exciting things you can do with your computer is to create colorful designs right on the screen. With your computer's graphic capability, you can make a design, draw a picture, create a gameboard, and so on.

In this chapter, we'll introduce you to two simple, yet important, graphics statements. CALL VCHAR and CALL HCHAR are used to position a character or draw a line of characters on the screen. Later chapters will show you how to choose and combine colors and how to use graphics statements in programs.

Earlier we mentioned that TI BASIC uses 28 printing positions on each line. For graphics, however, the computer allows 32 character positions on each line. Think of the screen as a "grid" of square blocks made up of 32 columns and 24 rows.

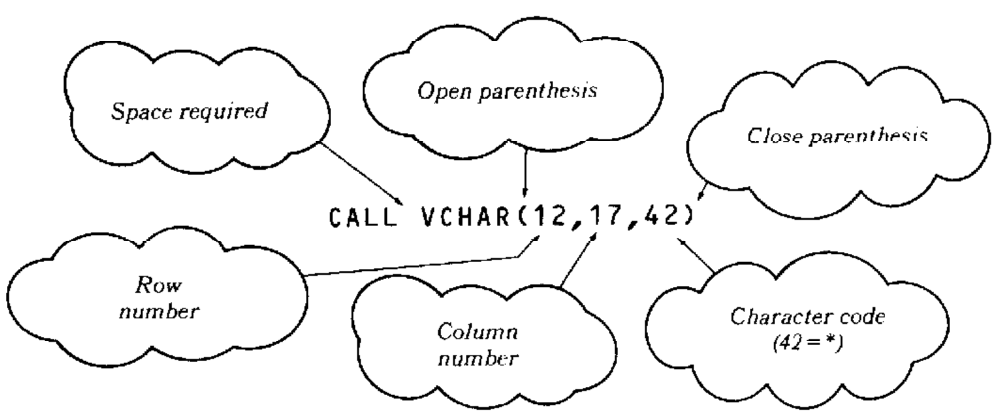


Each square on the grid is identified by two values (called *coordinates*) – a row number and a column number. For example, the coordinates 5,7 mean the fifth row and the seventh column, and the coordinates 10,11 mean the tenth row and the eleventh column.



The first thing we want to try is to place a character in a particular square on the screen. For the time being let's consider that a character is any one of the 26 letters of the alphabet, the numbers 0 through 9, and certain other symbols, like the asterisk (*), the plus and minus signs (+ and -), and the slash (/). (Later on, in Chapter 5, you'll learn more about how to define other characters for graphics.) Each character is assigned an identifying numeric value of its own, and the values for the full character set are given in *Appendix B*.

By using either CALL VCHAR or CALL HCHAR, naming the two coordinates (row and column), and identifying a character by its numeric value, you can place the character in any spot you choose. Here's the form used for these two statements:

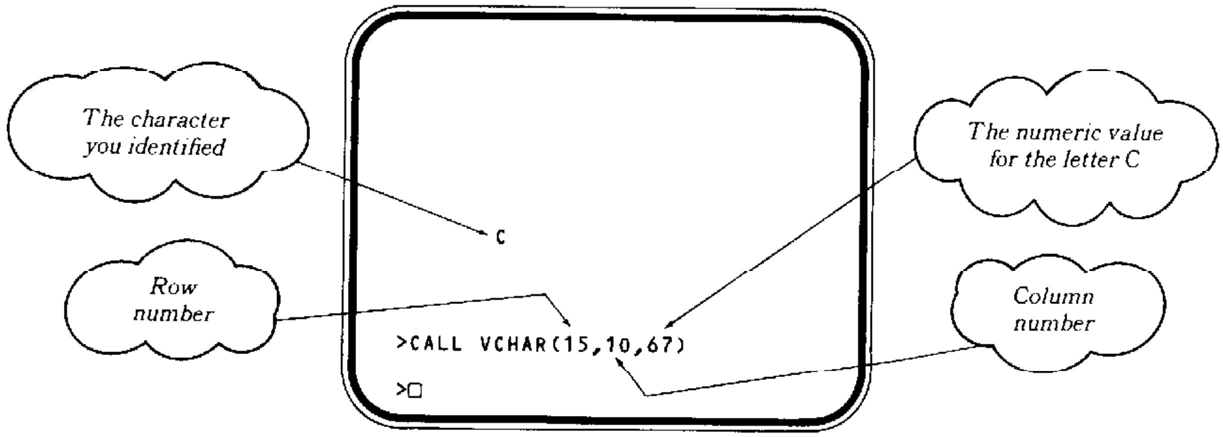


Try this example, and you'll see an asterisk (*) appear near the center of the screen.

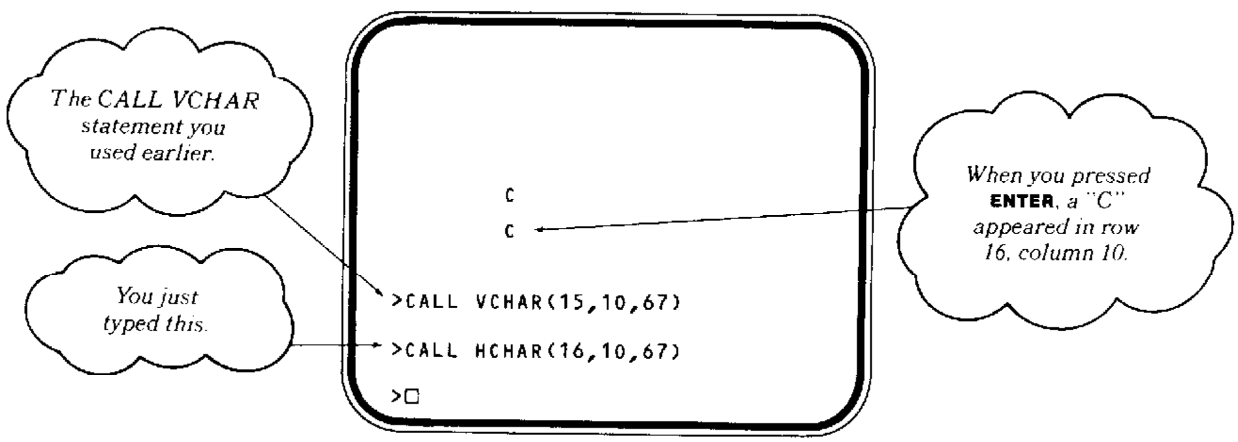
Let's try a few more examples. First, clear the screen by typing CALL CLEAR and pressing **ENTER**. Now type:

```
CALL VCHAR(15,10,67)
```

(Don't forget the parentheses in the statement – they're important!)



Now try the CALL HCHAR statement.



The order for entering the row number, the column number, and the character's numeric value is the same for both CALL VCHAR and CALL HCHAR, and they both do the same thing *when you are positioning a single character on the screen.*

If you want to draw a line of characters, however, you'll find that there is a distinct difference between the functions of the two statements. CALL VCHAR causes a *vertical* column of characters to appear, while CALL HCHAR draws a *horizontal* row of characters. To draw a line with either statement, we must add a fourth numeric value to the statement: the number of repetitions we want. This number controls the "length" of the line.

Clear the screen (type CALL CLEAR and press **ENTER**), and let's try a vertical line. Type this:

starting row number

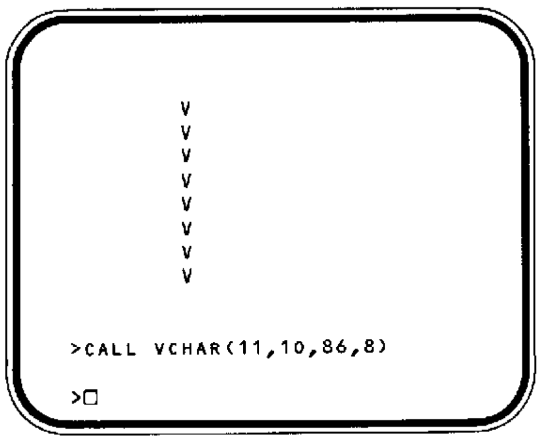
10th column

```
CALL VCHAR(11,10,86,8)
```

numeric value for "V"

number of repetitions

Check for errors, and then press **ENTER**. The screen will look like this:



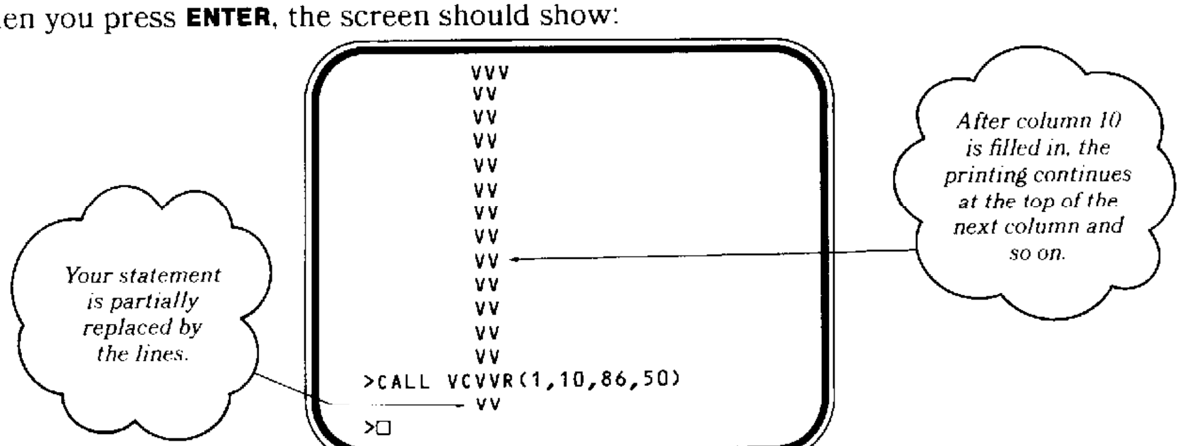
As we mentioned earlier, there are 24 horizontal rows of character blocks on the "grid" of the screen. Therefore, you can only draw a vertical line (column) that is 24 characters long. What will happen, then, if you enter a repeat value greater than 24? Let's try it.

Clear the screen and then type in:

```
CALL VCHAR(1,10,86,50)
```

We've asked for 50 repetitions this time.

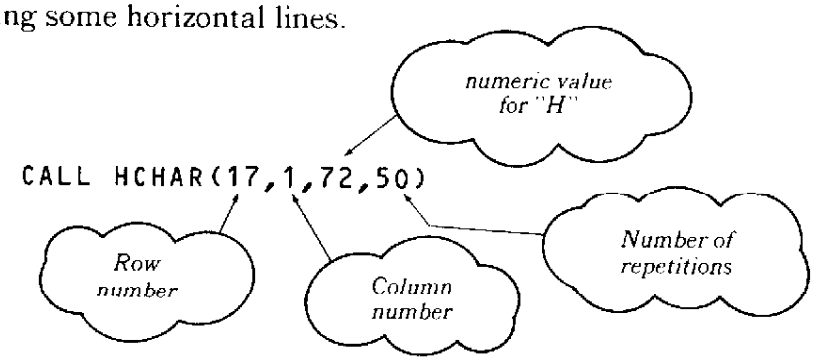
When you press **ENTER**, the screen should show:

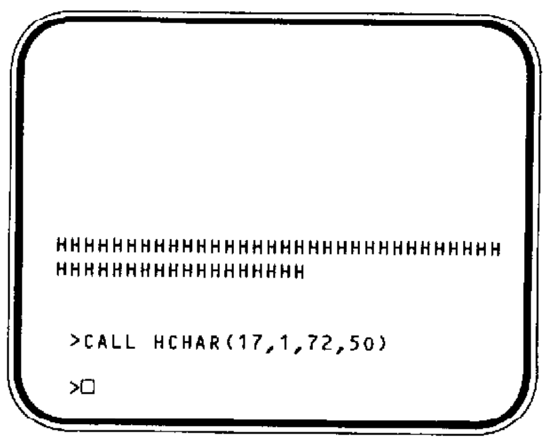


(*Note:* Graphics *in the Immediate Mode only* are affected by the scrolling of the screen. That's why you don't actually see all 50 of the V's above – some have already scrolled off the top of the screen.)

We also mentioned earlier that there are 32 vertical columns; therefore, it would seem that we could draw a horizontal line 32 characters long. However, some display screens may "clip off" the first two and last two columns (columns 1 and 2, 31 and 32). The only way to know what your screen shows is to experiment. So let's clear the screen and try drawing some horizontal lines.

Type in:





Again the printing filled one line (horizontal, this time) and then started over on the next line. Count the H's. If you see only 28 in the full line, columns 1 and 2, 31 and 32 do not show on your screen, and you should use only columns 3 through 30 to avoid losing part of your graphic design.

So far, we've entered actual numeric values in our statements. However, you can use the LET statement to assign numeric values to variables and then use the variables in the CALL VCHAR and CALL HCHAR statements. Try this:

```
LET A=5
LET B=12
LET C=67
CALL CLEAR
CALL VCHAR(A,B,C)
```

Where did the "C" appear on the screen?

Experiment!

For a big finale let's fill the screen with asterisks (numeric code 42). Type these lines, pressing **ENTER** at the end of each line.

```
CALL CLEAR
CALL HCHAR(1,1,42,768)
```

24 rows × 32 columns = 768 positions.

Continue to experiment on your own, trying different characters (see *Appendix B*) and positions. For example, can you fill the screen with your first-name initial?

SUMMARY OF CHAPTER 1

This concludes our "tour" in the Immediate Mode, and you've been introduced to these BASIC statements:

PRINT	CALL SOUND
CALL CLEAR	CALL VCHAR
LET	CALL HCHAR

This chapter has given you a glimpse of TI BASIC and your computer's capabilities. Now, you're ready to get into the real fun — learning to program your computer.

2 CHAPTER TWO — Simple Programming

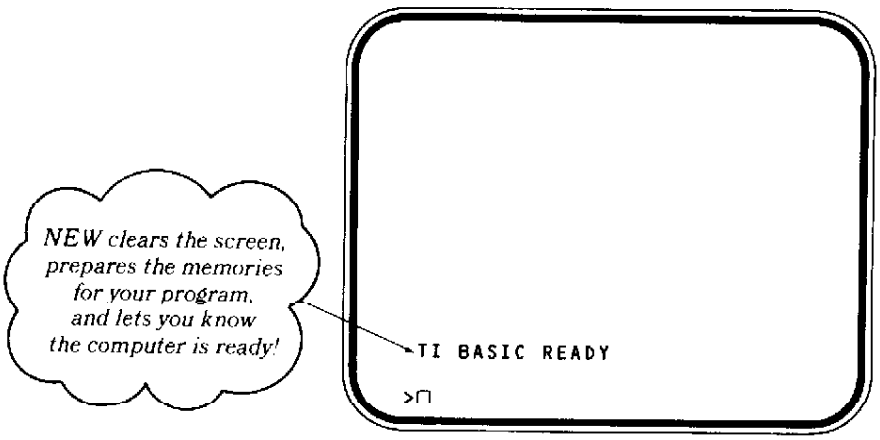
In Chapter 1, you used Immediate Mode statements to instruct the computer to do one thing at a time. Each statement was performed *immediately* after you pressed the **ENTER** key.

> You typed PRINT "HI THERE!" and pressed **ENTER**.
> The computer printed HI THERE!

Now you're ready to discuss *programs*, sets of statements which are *not* done immediately. Instead, they are stored in the computer's memory, waiting for you to instruct the computer to perform them.

A Printing Program

Let's begin by using an old familiar friend, the PRINT statement, in a program. First type the word NEW and press **ENTER**.



Now type the following program, pressing **ENTER** at the end of each program line:

```
10 PRINT "ARE YOU READY"
20 PRINT "TO LEARN BASIC?"
30 END
```

one space

(As you type the program, notice the small "prompting" character that appears just to the left of the printing area. This symbol marks the beginning of each program line you type.)

In computer terminology, you have just "entered" a program. Nothing to it! Check the program now to see if there are any typing mistakes. If there are, just retype the line correctly, *including the number at the beginning of the line*, right there at the bottom of the screen. Then press **ENTER**. The computer will automatically replace the old line with the new, correct one.

When you're ready to see the program in action, type CALL CLEAR and press **ENTER**. The screen will be cleared, but your program won't be erased – it's stored in the computer's memory!

Now type RUN and press **ENTER** again.

```
>RUN
 ARE YOU READY
 TO LEARN BASIC?

 ** DONE **

>□
```

Want to "run" the program again? Type RUN again and press **ENTER**.

```
>RUN
 ARE YOU READY
 TO LEARN BASIC?

 ** DONE **

>RUN
 ARE YOU READY
 TO LEARN BASIC?

 ** DONE **

>□
```

Each time you type RUN and press **ENTER**, the computer begins at the first statement and follows your instructions in order until it reaches the last statement. END means just what it says: the end, stop!

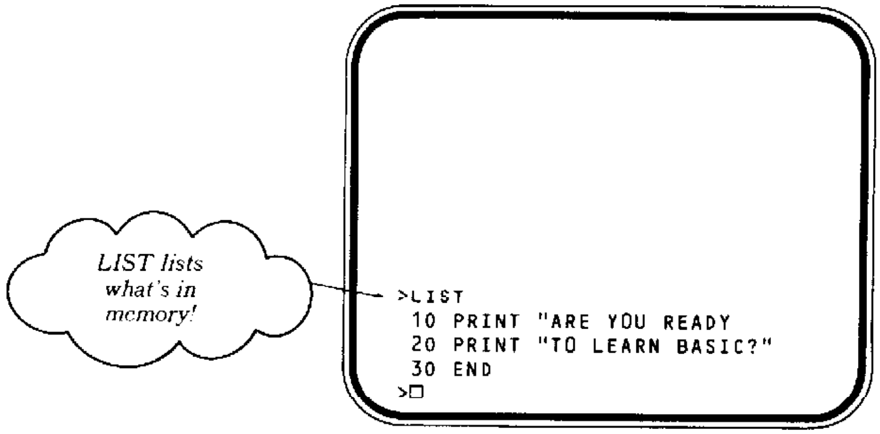
Did you notice that the display screen briefly turned green while the program was running? The screen always turns green while a program is being executed and then changes back to its normal blue color when the program is finished.

Program Structure

Now that you've had a bit of programming experience, let's review some of the things you did when you entered the program above. To refresh your memory, we'll get the program back on the screen.

2

First, type CALL CLEAR (*without* a line number) and press **ENTER** to clear the screen. Now type LIST and press **ENTER** again:



The program above consists of three statements or "lines." Each statement begins with a *line number*, which serves two important functions:

1. It tells the computer *not* to perform the statement immediately, but to store it in memory when you press **ENTER**.
2. It establishes the *order* in which the statements will be done in the program.

As in the Immediate Mode, you pressed **ENTER** when you finished typing each program line. Pressing **ENTER** defines the end of the program line, just as the line number identifies the beginning of the line. It is also the computer's cue to store the line in its memory. *Pressing* **ENTER** *at the end of each program line is essential* – without it, your line will not be correctly stored by the computer.

The computer performs the program in this order.

```
10 PRINT "ARE YOU READY"
20 PRINT "TO LEARN BASIC?"
30 END
```

Also, you may be wondering why we numbered the lines in increments of ten (10,20,30,etc.). Well, we could just as easily have numbered them 1,2,3. By using increments of ten, however, or other spreads like 100,200,300, etc., we can go back and insert new lines if we want to expand the existing program, and we don't have to retype the whole program! (We'll cover this clever trick when we discuss editing a program.)

Commands—NEW, RUN, LIST

You've already used these commands, but you might like a little more definition of commands in general and these three in particular at this point.

Commands are different from *statements*. They are not part of the program, and they do not have line numbers. Instead, they instruct the computer to do specific tasks:

NEW –Instructs the computer to erase the program in its memory. (It also clears the screen, but *don't* confuse it with CALL CLEAR, which *only* clears the screen.)

RUN –Instructs the computer to perform (or "run") the program in its memory.

LIST –Instructs the computer to show (or "list") on the screen the program that is stored in its memory.

As you saw earlier, we use NEW *only* when we want to prepare the computer for storing a new program. Be careful in using NEW; when in doubt, use LIST first, so that you can see the current program before you erase it.

LIST is a powerful aid for correcting or changing a program. It lets you get the program right on the screen in front of you, where you can check for and correct any errors in your program.

And you already know what RUN will do! It's the magic word that makes it all happen.

A Numerical Program

In addition to its printing or "message" capabilities, your computer also has a great deal of "number power." You experimented with addition, subtraction, multiplication, and division in the Immediate Mode in Chapter 1. Now it's time to try a mathematical problem-solving program. Just to refresh your memory, review the keys that are used to perform the four basic mathematical operations:

SHIFT + for addition
SHIFT – for subtraction
SHIFT * for multiplication
/ for division

*Notice that +, –, and * are the upper symbols above =, /, and 8.*

As an example, we can easily construct a program that will convert kilograms to pounds (1 kilogram=2.2 pounds). The first thing we'll do is to clear the display and the computer's memories by typing NEW and pressing **ENTER**. We'll use the variables K (for kilograms) and P (for pounds) to help us remember which value is which, and we'll begin our program by assigning values to these variables.

Type:

```
10 LET K=50
20 LET P=2.2*K
```

Press **ENTER**.

In this case, we are trying to find out how many pounds are equal to 50 kilograms, so we have defined K as 50. Notice that we have defined P as 2.2×K. If we stopped here and ran the program at this point, the computer would perform the conversion, but it wouldn't show us the answer! So type in:

```
30 PRINT P
```

and press **ENTER**. Now, have we told the computer everything it needs to do? We've told it the number of kilograms we want converted to pounds, we've told it how to make the conversion, and we've told it to show us the answer. Yes, that's all we need, so type:

```
40 END
```

and press **ENTER**. Your program should look like this:

```
TI BASIC READY

>10 LET K=50
>20 LET P=2.2*K
>30 PRINT P
>40 END
>□
```

Before you run the program, let's mention two features of TI BASIC that may be slightly different from other versions of the language. First, a prompting character (to the left of the printing field on the screen) marks the start of every program line you type. You'll see its function more clearly when you begin to enter program lines that are longer than a single screen line. Second, the END statement in a program is optional in TI BASIC. Since it is a conventional part of BASIC, however, we'll use it in this example.

Now check the program for typographical errors. If there are any, retype the line correctly, including the line number, and press **ENTER**. When you're ready, type RUN and press **ENTER**.

```
TI BASIC READY

>10 LET K=50
>20 LET P=2.2*K
>30 PRINT P
>40 END
>RUN
 110

 ** DONE **

>□
```

the answer

Your answer is on the screen: 50 kilograms is equal to 110 pounds. Suppose, however, that we want to find the number of pounds that are equivalent to 60 kilograms. Easy! We can do it by changing only one line – line 10. Type:

```
10 LET K=60
```

and press **ENTER**. Now type RUN and press **ENTER** again.

your first answer, P=2.2×50

your second answer, P=2.2×60

```
>10 LET K=50
>20 LET P=2.2*K
>30 PRINT P
>40 END
>RUN
 110

 ** DONE **

>10 LET K=60
>RUN
 132

 ** DONE **

>□
```

Editing the Program

What you have just done is called "editing" a program. The ability to edit or change a program without retyping the whole thing is one you'll come to value highly as your programming skills grow. To give you an idea of the great flexibility editing adds to programming, let's experiment with a few more changes in the present program.

Adding Program Lines

We mentioned earlier that the reason we number program lines in increments of 10 (instead of 1,2,3, etc.) is to allow program lines to be added without retyping the whole program. Before we experiment with a few examples, let's clear the screen and recall our program.

Type:
```
CALL CLEAR
LIST
```
Press **ENTER**.

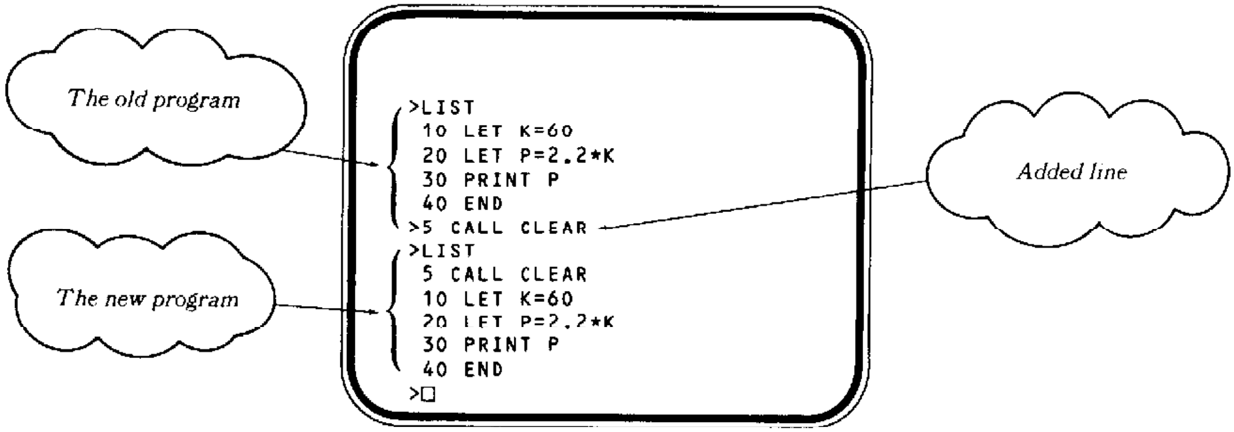
```
>LIST
 10 LET K=60
 20 LET P=2.2*K
 30 PRINT P
 40 END
>□
```

(Notice that the prompting character doesn't appear to the left of lines printed by the computer — only the lines you type are marked!)

We might want to add a CALL CLEAR statement to the program, so that we won't have to keep clearing the screen from the keyboard each time we "run" the program.

Type:
```
5 CALL CLEAR
```
Press **ENTER**.

Now list the program again to see the new line (type LIST and press **ENTER**).



Compare the two programs on the screen, and notice that the computer has automatically placed the new line in its proper order. Run the program again to see the effect of the added line.

Now let's add another line that will help to point out our answer. Type:

```
27 PRINT "THE ANSWER IS:"
```

and press **ENTER**. When you run the program again, you'll see this:

```
THE ANSWER IS:
 132

** DONE **

>□
```

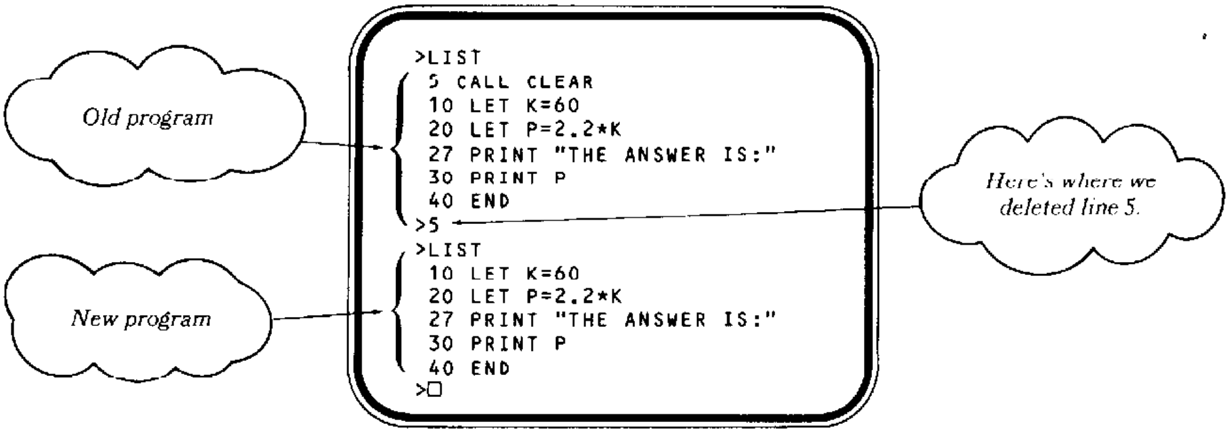
Removing Program Lines

Quite often it's necessary to remove a line or lines from a program. Deleting a program line is a very simple procedure.

The program we have stored right now doesn't really have any lines we want to delete. Just for practice, however, let's remove line 5.

First, clear the screen and list the program as it is now. Line 5 is the first line of the program, a CALL CLEAR statement. To remove it, simply type **5** and press **ENTER**.

Then list the program again. Presto! Line 5 is gone!



That's all there is to it. To remove a line, type the line number and press **ENTER**. The computer will then delete the line from program memory.

Since we really need line 5 in this program, let's reenter it. Type

```
5 CALL CLEAR
```

and press **ENTER**.

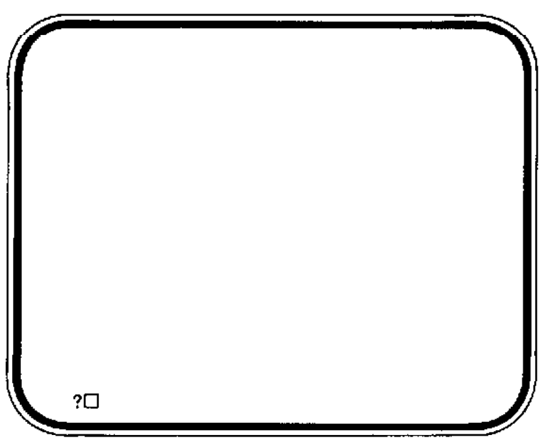
The INPUT Statement

You've already seen that you can easily change the value of K by simply retyping line 10 to assign a new value. But suppose you had many values for K, and you wanted to find the equivalent value of P for each one. It would get rather tiresome to retype line 10 each time.

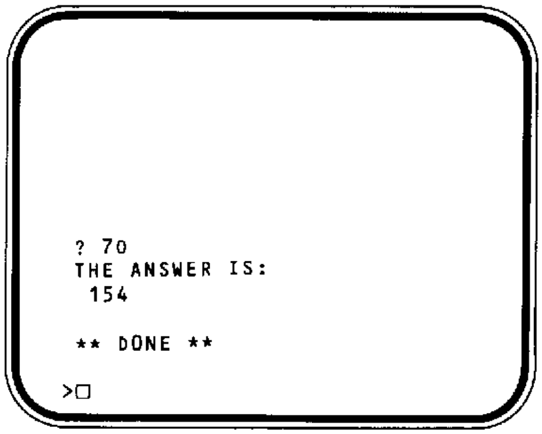
There is a better way to edit line 10. An INPUT statement causes the computer to type a question mark and stop, waiting for you to type in a value and press **ENTER**. The value you enter is then assigned to the variable contained in the INPUT statement. For example, type

```
10 INPUT K
```

and press **ENTER**. Now run the program again.



The question mark and cursor show you that the computer is waiting for you to "input" a value for K. This time we'll let K=70, so type *70* and press **ENTER**. The computer prints your answer:



Now you can run the program as many times as you like, changing the value of K each time the computer prints a question mark and stops. Try the program several times with different values for K.

The INPUT statement can also be used to print a "prompting" message (instead of a question mark) that helps you remember what value the computer is asking for. Change line 10 again by typing

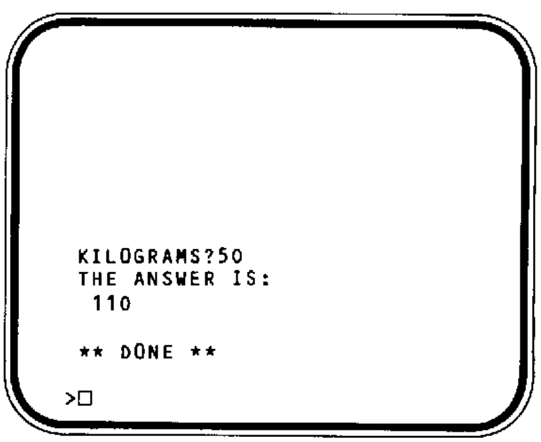
```
10 INPUT "KILOGRAMS?":K
```

colon

and pressing **ENTER**. Now run the program again. First the program asks:

```
KILOGRAMS?
```

Let's let K=50 this time. Type *50* and press **ENTER**.



By now, your program looks like this:

```
5 CALL CLEAR
10 INPUT "KILOGRAMS?":K
20 LET P=2.2*K
27 PRINT "THE ANSWER IS:"
30 PRINT P
40 END
```

If you'd like, you can list it on the screen at this time and review the changes you've made so far. When you're ready, we'll go on to look at one more change.

String Variables

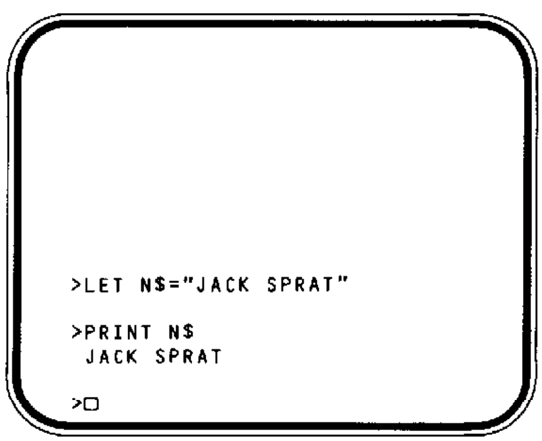
You already know what *numeric variables* are: numeric values assigned to names (variables), like "K=50." A string variable is a combination of characters (letters and numbers, or other symbols) assigned to a name. String variables differ from numeric variables in these ways:

1. The variable name *must* end with a $.
2. The alphanumeric characters in the "string" *must* be enclosed in quotation marks.
3. "Strings" of numbers cannot have arithmetic operations performed with or upon them.

Let's try a couple of examples in the Immediate Mode before changing the program. (Note that this does not interfere with the program stored in memory!)

Clear the screen (CALL CLEAR) and enter this:

```
LET N$="JACK SPRAT"

PRINT N$
```



Now type:

one space here

```
LET W$=" ATE NO FAT."

PRINT N$;W$
```

semicolon

```
>LET N$="JACK SPRAT"

>PRINT N$
 JACK SPRAT

>LET W$=" ATE NO FAT."

>PRINT N$;W$
 JACK SPRAT ATE NO FAT.

>□
```

Let's make your conversion program a little more personal by using a string variable. Type these two lines:

colon

```
8 INPUT "NAME, PLEASE?":B$
26 PRINT "OK, ";B$
```

semicolon

leave one space

(Clear the screen and list the program again so you can see how the new lines fit in.)

When you run the program this time, the two INPUT statements will stop the program twice:

The computer asks	*You type in*
NAME, PLEASE?□	Your name and then press **ENTER**.
KILOGRAMS?□	The number of kilograms and then press **ENTER**.

Let's try it. Type RUN and press **ENTER**.

```
NAME, PLEASE?□
```

We'll type in Alpha (that's a nice name) and press **ENTER**. Then we'll see

```
NAME, PLEASE?ALPHA
KILOGRAMS?□
```

Again let's type *70* for the number of kilograms. Press **ENTER** again and you'll see:

```
NAME, PLEASE?ALPHA
KILOGRAMS?70
OK,ALPHA
THE ANSWER IS:
 154

** DONE **

>□
```

String variables can save a lot of typing when you're using a message (a name or a prompting statement, for example) more than once in a program.

Now list your program and review these latest changes. We've given you a lot of information, and we've given it pretty quickly. This would be a good time for you to do a little experimenting on your own, trying out some of the things you've learned.

Experiment!

Want a challenge? Try writing another conversion program — one that converts a temperature in degrees Fahrenheit (F) to degrees Celsius (C). The conversion formula is

Degrees C = 5/9 (Degrees F − 32)

Don't forget to use INPUT statements and CALL CLEAR at appropriate places! Hint: Let C = 5/9*(F − 32) — the parentheses must be there in your program!

2

The GO TO Statement

So far, you've been developing programs that operate from beginning to end in a straight sequential order. There are many situations, however, in which you want to interrupt this orderly flow of operation. Look at the following program, but don't enter it yet:

```
10 CALL CLEAR
20 INPUT K
30 PRINT K
40 PRINT
50 K=K+1
60 GO TO 30
```

Here we "send" the program back to line 30 by using a GO TO statement in line 60. The GO TO statement causes the actions performed by lines 30, 40, and 50 to be repeated over and over again, setting up what's called a *loop.* (Notice that we don't use an END statement. That's because the program will never get beyond line 60! It won't stop until you tell it to by pressing **CLEAR.** This is called an "endless loop.")

Let's enter the program now. First, type NEW and press **ENTER** to erase the computer's memory, and then type these lines:

```
10 CALL CLEAR
20 INPUT K
30 PRINT K
40 PRINT
50 K=K+1
60 GO TO 30
```

Note that LET is optional.

Before you run the program, we'll examine a diagram called a *flowchart,* explaining how the program works.

Program Line	*Operation*
10 CALL CLEAR	Clears the screen
20 INPUT K	Stops and waits for initial value of K
30 PRINT K	Prints the current value of K
40 PRINT	Prints nothing; just gives you a blank line
50 K=K+1	Reassigns a new value to K (the old value + 1)
60 GO TO 30	Transfers the program back to line 30

Now run the program, putting in 1 for the beginning value of K. Watch how quickly the computer counts — almost too fast to follow! That's why we added the blank line (line 40). This line spaces out the numbers a bit so that you can see them better.

Let the computer count as long as you want to. When you are ready to stop the program, press **CLEAR.** You'll see *BREAKPOINT AT (#) on the screen, indicating where the program stopped. Run the program as many times as you want, using whatever number you wish as the initial value for K (50,100,500,etc.).

GO TO can be typed as GOTO in your program. The computer isn't fussy about that. If you try to send the program to a non-existent line number, however, you'll get an error message.

Suppose, for example, we type in

```
60 GO TO 25
```

and press **ENTER**. Try it, run the program, and see what happens! You'll see this error message:

```
* BAD LINE NUMBER IN 60
```

So correct the line by typing and entering

```
60 GO TO 30
```

and run the program again.

Can we change the program to make it count by 2's, or 5's? You bet we can! By making one program change, let's make the computer count by 2's: Type:

```
50 K=K+2
```

and press **ENTER**. Now run the program, typing in 2 when the computer asks for the starting value of K.

Experiment with the program for a while, making it count by 3's, 5's, 10's, etc.

A GO TO Loop With CALL SOUND

GO TO *loops* have many applications, of course, beyond simple counting. We could use a loop, for example, to practice a musical scale.

Before we start the program, you might want to review the CALL SOUND section in Chapter 1 (see page 17) to help you remember how the CALL SOUND statement performs in the Immediate Mode. (It behaves essentially the same way in a program.)

When you're ready to start the program, type NEW and press **ENTER**. Our first task in the program will be to assign values to the variables we'll use. Type these lines:

```
10 LET T=300
20 LET V=2
30 C=262
40 D=294
50 E=330
60 F=349
70 G=392
80 A=440
90 B=494
100 HIC=523
```

"T" for "time"

"V" for "volume"

This is a C-scale.

frequency of middle C on the piano – note that the word LET is optional.

"HIC" for "high C"

Now you're ready for the CALL SOUND statements to tell the computer when to play each note:

```
200 CALL SOUND(T,C,V)
300 CALL SOUND(T,D,V)
400 CALL SOUND(T,E,V)
500 CALL SOUND(T,F,V)
600 CALL SOUND(T,G,V)
700 CALL SOUND(T,A,V)
800 CALL SOUND(T,B,V)
900 CALL SOUND(T,HIC,V)
```

Finally, set up the loop with a GO TO statement:

```
950 GOTO 200
```

Check the program now for errors, and correct any that you find. When everything is correct, run the program. Again, this is an endless loop. (Notice that the screen background stays light green until you stop the program.) You'll have to press **CLEAR** to stop it.

Experiment!

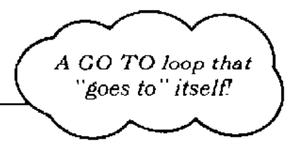
Practice building other musical scales and patterns, using the note frequencies listed in *Appendix A*.

A GO TO Loop with CALL COLOR

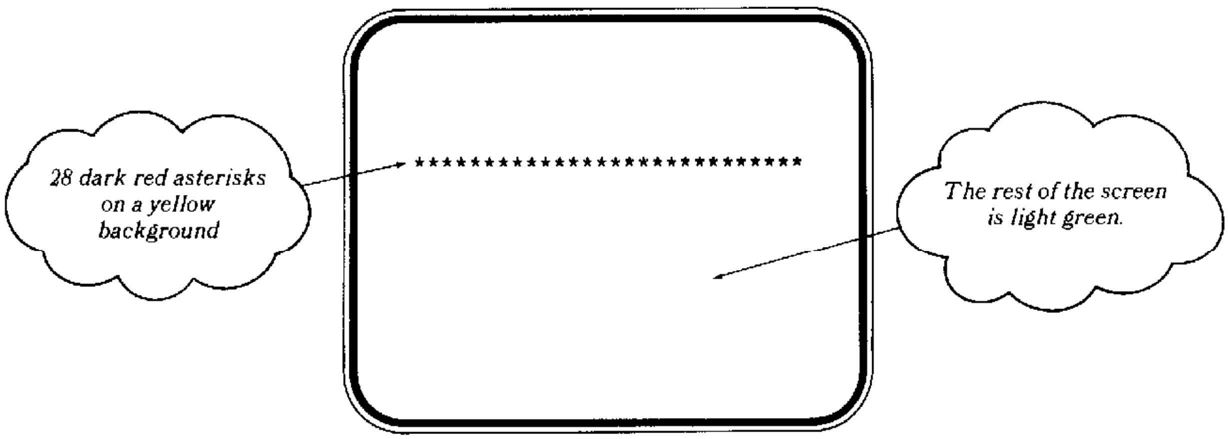
Up to now, you've seen only three colors in BASIC on your display screen. (Maybe you've only noticed two – but there really are three.) First, while you're entering a program, the screen background is cyan (a light blue color), and the characters (letters and numbers) that you're typing are black. Then, while the program is running, the screen becomes a light green color. When the program stops, the screen returns to cyan with black characters.

These are only three of the sixteen colors available with your computer, and the way you control the colors in a program is through the CALL COLOR statement. Let's try a program with a CALL COLOR statement and a slightly different GO TO loop. Clear your old program from the computer's memory (NEW; press **ENTER**), and type these lines:

```
10 CALL CLEAR
20 CALL COLOR(2,7,12)
30 CALL HCHAR(12,3,42,28)
40 GO TO 40
```



Now run the program, and the screen should look like this:



Our program prints twenty-eight asterisks across the screen. The asterisks are dark red. In the area where the asterisks are displayed, the screen color is a light yellow. The rest of the screen remains light green.

(Line 40 puts your program into a holding pattern that keeps your graphic on the screen. When you're ready to stop the program, press **CLEAR** to break the loop. Remember, you can run the program as many times as you like.)

A CALL COLOR statement requires three numbers, enclosed in parentheses and separated by commas:

```
20 CALL COLOR(2,7,12)
```

The first number after the open parenthesis symbol is a *character set number.* As we mentioned in Chapter 1, each *character* (letters, numbers, and symbols) that prints on the screen has its own numeric code, ranging from 32 through 127 for a total of ninety-six characters. These characters are organized by the computer into twelve *sets* with eight characters in each:

2

Set #1		Set #2		Set #3		Set #4	
Code	*Character*	*Code*	*Character*	*Code*	*Character*	*Code*	*Character*
32	(space)	40	(	48	0	56	8
33	!	41	)	49	1	57	9
34	"	42	*	50	2	58	:
35	#	43	+	51	3	59	;
36	$	44	,	52	4	60	<
37	%	45	-	53	5	61	=
38	&	46	.	54	6	62	>
39	'	47	/	55	7	63	?

Set #5		Set #6		Set #7		Set #8	
Code	*Character*	*Code*	*Character*	*Code*	*Character*	*Code*	*Character*
64	@	72	H	80	P	88	X
65	A	73	I	81	Q	89	Y
66	B	74	J	82	R	90	Z
67	C	75	K	83	S	91	[
68	D	76	L	84	T	92	\
69	E	77	M	85	U	93	]
70	F	78	N	86	V	94	^
71	G	79	O	87	W	95	_

Set #9		Set #10		Set #11		Set #12	
Code	*Character*	*Code*	*Character*	*Code*	*Character*	*Code*	*Character*
96	`	104	H	112	P	120	X
97	A	105	I	113	Q	121	Y
98	B	106	J	114	R	122	Z
99	C	107	K	115	S	123	{
100	D	108	L	116	T	124	¦
101	E	109	M	117	U	125	}
102	F	110	N	118	V	126	~
103	G	111	O	119	W	127	DEL

The set number you use in a CALL COLOR statement, then, is determined by the character you want to print. (And what happens if you want to print characters from different sets in the same colors? We'll discuss that in a few minutes.)

The second and third numbers in parentheses determine the colors used in your graphic. Each of the sixteen colors has its own numeric code.

Color	*Code #*	*Color*	*Code #*
Transparent	1	Medium Red	9
Black	2	Light Red	10
Medium Green	3	Dark Yellow	11
Light Green	4	Light Yellow	12
Dark Blue	5	Dark Green	13
Light Blue	6	Magenta	14
Dark Red	7	Gray	15
Cyan	8	White	16

The second number sets the *foreground* color; that is, the color of the character you designate. The third number sets the *background* color – the color of the block or square in which the character is printed.

Foreground color – Dark Red

```
20 CALL COLOR(2,7,12)
```

Background color – Light Yellow

The next line in your program is

```
30 CALL HCHAR(12,3,42,28)
```

row number
column number
numeric code for *
number of repetitions

(If you need to review the CALL HCHAR examples in Chapter 1, this would be a good time to do it.)

Now you know why we indicated Set #2 in our CALL COLOR statement! The asterisk (code number 42) is a part of Set #2.

Line 40 of the program is a GO TO statement that "goes to" itself. It keeps the computer "idling" until you press **CLEAR.** When you do, the program stops, and the screen changes back to its normal color. All the reds, yellows, and greens disappear.

Now let's change line 20 of the program to see some new colors. Stop the program, if it's still running, and type this:

```
20 CALL COLOR(2,12,5)
```

same set number
light yellow
dark blue

Press **ENTER** to store your new line, and list the program (LIST; press **ENTER**) to review your program.

When you're ready, run the program. You'll see 28 light yellow asterisks against a dark blue background this time.

You could, of course, continue to experiment by stopping the program, entering a new line 20 and running the modified program over and over. Don't. Instead, save wear and tear on your fingers by entering the following program which allows you to experiment more easily. With this program, you enter foreground (F) and background (B) colors in response to INPUT statements.

INPUT color codes.

```
NEW
10 CALL CLEAR
20 INPUT "FOREGROUND?":F
30 INPUT "BACKGROUND?":B
40 CALL COLOR(2,F,B)
50 CALL HCHAR(12,3,42,28)
60 GO TO 60
```

Set colors based on INPUT color codes.

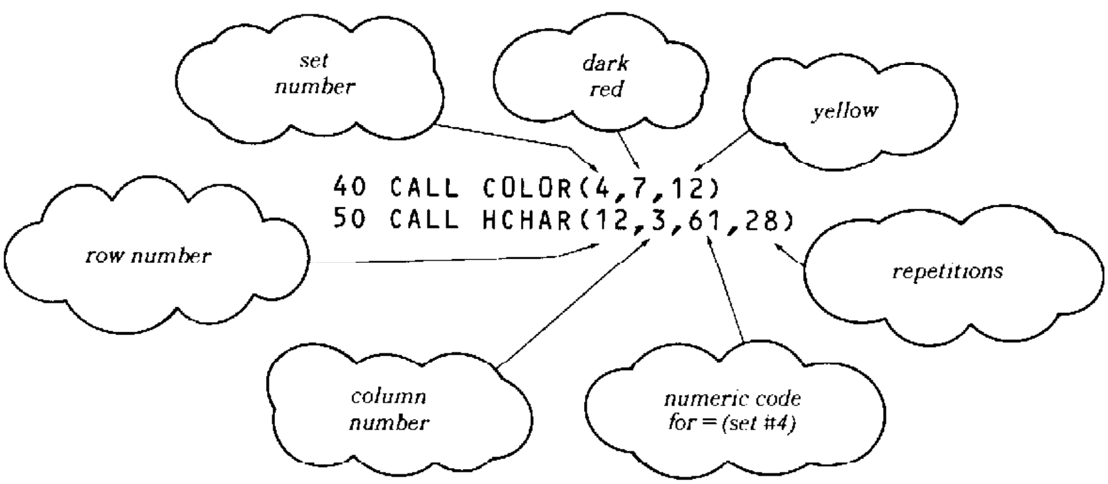
You know what this does.

When the computer asks you for the "foreground" and "background" colors you want to use, you can type in any color code from *1* through *16*. Remember, however, that color number 1 is transparent and color 4 is the screen color when the program is running. These may not be satisfactory in this program. (Also, color number 2, black, can cause display distortion on some screens.) Here are some combinations you might find interesting:

Foreground Color	*Background Color*
3	16
2	11
5	16
2	14
7	15
7	12
13	12
14	16

OK, have you checked your program for typographical errors? Have you chosen the foreground and background colors you want to use first? Then run the program.

After you've experimented with different color combinations, you might enjoy trying some other characters, as well. You can do this by retyping line 50, substituting a different character code number for the "42" (asterisk code number). Just remember, if you select a character from any set other than #2, you'll also have to change line 40 to reflect the new set number. For example:



Earlier we brought up this subject: What happens if you want to print characters from different sets, all in the same color? One way to do this is to include in your program twelve CALL COLOR statements – one for each set of characters. You'll have to do quite a bit of typing, *but* you'll be free to use any of the characters you choose. Try the following program, which covers eight character sets.

```
NEW
100 CALL CLEAR
110 CALL COLOR(1,6,16)
120 CALL COLOR(2,6,16)
130 CALL COLOR(3,6,16)
140 CALL COLOR(4,6,16)
150 CALL COLOR(5,6,16)
160 CALL COLOR(6,6,16)
170 CALL COLOR(7,6,16)
180 CALL COLOR(8,6,16)
190 PRINT "              "
200 GOTO 200
```

Light blue

White

You decide what to do here!

Use any message you want in line 190; just remember to enclose it in quotation marks. With these CALL COLOR statements you have told the computer to print any of the sixty-four characters in light blue (6) on a white (16) background.

Experiment!

Put a little COLOR in your life! Try some experiments of your own with different colors and character sets. For example, what happens if you enter the same color code for foreground and background? Try it!

Error Messages

We haven't talked much in this chapter about error messages because, for the most part, the ones you'd run into in these program examples are the same as – or very similar to – those you learned about in Chapter 1. For example, a spelling or typing error in NEW, RUN, or LIST will cause the computer to return an "INCORRECT STATEMENT" message as soon as you press **ENTER**.

Errors in program statements may be detected by the computer either when the line is entered *or* when the program is run. Here is a sample of error conditions and messages you might see when you *enter* an incorrect line:

Condition	*Message*
Omitting a quotation mark: `10 INPUT "WHAT COLOR:F`	`* INCORRECT STATEMENT`

Below are some examples of line errors that would cause error messages when you *run* a program:

Condition	*Message*

Misspelling a statement:
`10 INPT "WHAT COLOR":F`
Omitting necessary punctuation or typing an incorrect punctuation mark:
`10 INPUT "WHAT COLOR"F`
`10 INPUT "WHAT COLOR";F`
	`* INCORRECT STATEMENT IN 10`

2

Leaving the variable out of an INPUT statement:

```
10 INPUT "WHAT COLOR":
```

Leaving out the space between GO TO and the line number:

```
10 GO TO30
```

```
* INCORRECT STATEMENT IN 10
```

Using a non-existent line number in a GO TO statement:

```
10 LET A=5
20 PRINT A
30 GO TO 15
```

?? There is no line 15!

```
* BAD LINE NUMBER IN 30
```

Notice that the error messages given during a program run usually indicate the number of the troublesome line. If you'd like to view the line in question (let's say it's line 10), just type

```
LIST 10
```

and press **ENTER**. The computer will obediently print line 10 on the screen for you to review. You can also list the whole program on the screen if you prefer. Type

```
LIST
```

and press **ENTER**.

Remember, too, that failing to press **ENTER** at the end of each program line may cause the computer to give you an error message or an incorrect result, depending on the kind of operation you're performing.

Making mistakes is a normal part of learning – so don't be disturbed when the computer gives you an error message. Just list the line or the program, identify the error, retype the line correctly, and go right on your way!

(*Note:* If you'd like to see all the error messages your computer can give you, or if you don't understand a message you're given, you'll find a complete list of error messages – and when they occur – in the "BASIC Reference Section" of your *User's Reference Guide.*)

SUMMARY OF CHAPTER 2

In this chapter you've covered a lot of very important ground. You've learned how to:

- Enter a program
- Use the commands NEW, LIST, and RUN
- "Edit" or change a program
- Use INPUT statements with *numeric variables* and *string variables*
- Build a mathematical conversion program
- Create a GO TO loop within a program
- Stop an endless loop with **CLEAR**
- Use a GO TO loop in a CALL SOUND program
- Use the CALL COLOR statement and a GO TO loop in a graphics program

When you started working with Chapter 2, you were a beginner in learning BASIC and programming. Now you're well on your way to becoming an experienced computer programmer.

Quick Review of Program Structure

1. Begin each line with an identifying line number (1-32767).
2. Number the lines in the order you want the computer to follow in performing the program.
3. Press **ENTER** when you have finished typing a program line.

3 CHAPTER THREE
More Programming Power

By this time you've had quite a bit of programming experience in TI BASIC. You know what a program is, how it's structured, and how it's performed by the computer. Now we're ready to add a few more techniques to your programming skills.

In this chapter we'll introduce you to several new TI BASIC features. First, there's the very useful and versatile FOR-NEXT statement, which creates loops in programs. Next, we'll cover some "plain and fancy" printing, using the PRINT statement and the TAB function. Then we'll add some details about the "number power" of your computer: the way numbers are displayed on the screen and the order in which the computer performs mathematical calculations. Finally, we'll introduce you to the INTeger function.

These new features will help you increase your programming skills, building on those we've already discussed in previous chapters. They'll also prepare the way for even more exciting things to come.

The FOR-NEXT Statement

Chapter 2 presented several examples of the GO TO loop, which repeats a set of statements indefinitely – or until you press **CLEAR** to stop the program. The FOR–NEXT statement also creates a loop, but it's different from GO TO in two important ways:

1. The FOR-NEXT statement is actually a pair of lines in the program, the FOR line and the NEXT line, each with its own line number.
2. *You* control the number of times the loop is performed. After the loop has been "executed" the number of times you specify, the program moves on to the line that follows the NEXT line.

The FOR line has the form

space required

upper limit

variable

```
30 FOR A=1 TO 3
```

starting count

The NEXT line could be

space

```
80 NEXT A
```

the same variable used in the FOR line

These two lines would cause the portion of the program between the FOR and NEXT lines to be performed three times. In this example the starting value of A is 1; after each pass through the loop, A is increased by 1. Its value is then tested against the upper limit (3, in this example). After the third pass through the loop, A is equal to 4, so the program "exits" (or leaves) the loop to the line following line 80.

To help you see the differences between GO TO and FOR-NEXT more clearly, let's compare two similar programs, one with a GO TO loop and one with a FOR-NEXT loop.

A GO TO Loop

Type NEW, press **ENTER**, and then enter this program:

```
10 CALL CLEAR
20 LET A=1
30 PRINT "A=";A
40 LET A=A+1
50 GO TO 30
```

Before you run the program, think for a few minutes about what it will do. First, the initial value of the variable A will be set to 1. Then, the computer will print out the current value of A. Finally, the value of A will be increased by 1, and the program will loop back to line 30. It will go on with this procedure until you press **CLEAR.**

Ready to run the program? Type RUN and press **ENTER** to see it in action. When you're ready to stop it, press **CLEAR.**

A FOR-NEXT Loop

Now let's examine a similar "counting" program with a FOR-NEXT loop. Type NEW and press **ENTER** to erase the first program. Then type these lines:

```
10 CALL CLEAR
20 FOR A=1 TO 5
30 PRINT "A=";A
40 NEXT A
50 PRINT "OUT OF LOOP"
60 PRINT "A=";A
70 END
```

Think about the way this program will be performed. The value of A will start at 1 and will be increased by 1 each time the program completes line 40. As soon as the value of A is *greater than* 5, the program will exit the loop and continue with line 50. If we listed the lines in their order of performance, along with the increasing values of A, this is what we'd have:

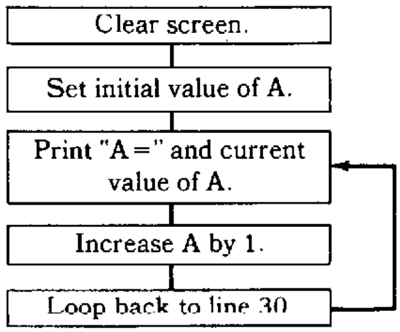
	Line Number	Value of A
Sets up value limits for A and starts loop.	10	0
	20	1
First pass through loop	30	1
	40	2
Second pass	30	2
	40	3
Third pass	30	3
	40	4
Fourth pass	30	4
	40	5
Fifth pass	30	5
	40	6
Out of loop	50	6
	60	6
	70	6

Run the program, and the screen should look like this:

```
A= 1
A= 2
A= 3
A= 4
A= 5
OUT OF LOOP
A= 6

** DONE **

>□
```

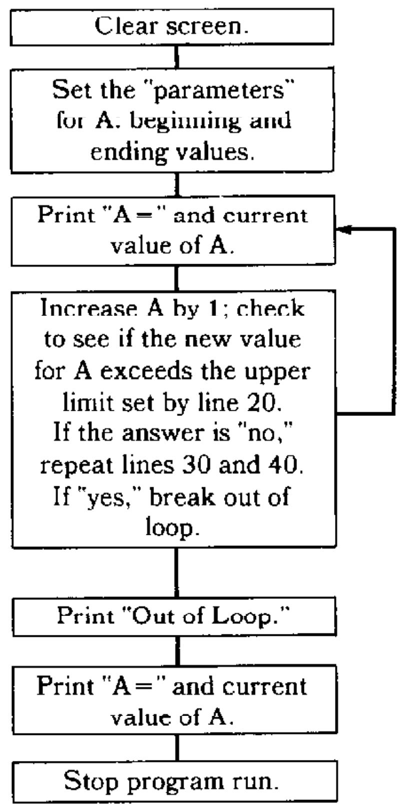
The following flowcharts illustrate the differences in the two programs.

GO TO Program



(Loop continues until you stop the program by pressing **CLEAR**.)

FOR-NEXT Program



In Chapter 2 we also used the GO TO statement in a CALL COLOR program to create a *delay loop:*

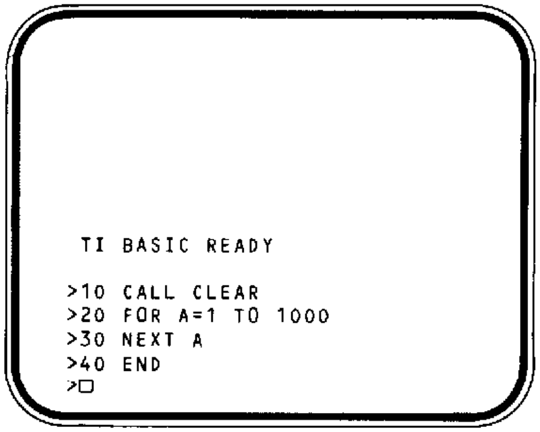
```
40 GO TO 40
```

This line caused the program to "idle" and hold the color design on the screen until you pressed **CLEAR.** Without some sort of delay loop, the color we used in the program would have blinked on the screen only for an instant before the program stopped and the screen returned to its normal Immediate Mode colors.

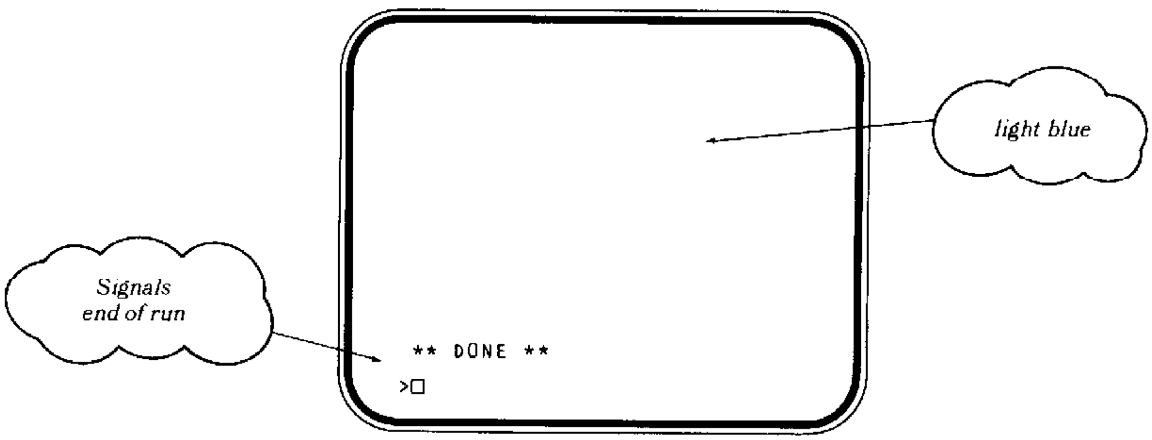
We can also use the FOR-NEXT statement to build a controlled time delay into a program. Consider this example:

```
20 FOR A=1 TO 1000
30 NEXT A
```

Better still, let's try it! Type NEW, press **ENTER**, and then type in the following program:



Now run the program. What happens on the screen? Not much, really; the screen changes to a light green, and the cursor disappears. After a short time delay (while the computer "counts" from 1 to 1000), the screen changes back to cyan (a light blue) and the cursor reappears:



Although no other lines are being executed between the FOR and NEXT lines, time passes while the computer counts the number of loops, in this example from 1 to 1000. The following program utilizes a FOR-NEXT time-delay loop in a CALL COLOR program.

CALL COLOR with a FOR-NEXT Loop

Clear the previous program (type NEW; press **ENTER**), and enter this program:

```
10 CALL CLEAR
20 CALL COLOR(2,7,7)
30 CALL HCHAR(12,3,42,28)
40 FOR B=1 TO 1000
50 NEXT B
60 END
```

Color codes for foreground and background are the same: dark red

This program will print a row of asterisks on the screen. However, since the *foreground* color (the color of the asterisks) and the *background* color are both dark red, the screen will show a solid horizontal bar of dark red. The red asterisks blend into the red backgrounds.

Now run the program. Does the color bar stay on the screen long enough for you to observe it carefully? If not, change line 40 to increase the time delay (1 to 2000, for example).

Suppose we want to see a bar of a different color? We could retype line 20, inserting a new color code for the foreground and background colors. But there's an easier way to edit the program so that we won't have to retype line 20 every time we want to change colors. Type these lines:

```
15 INPUT A
20 CALL COLOR(2,A,A)
60 GO TO 10
```

Here's where we'll enter our new color code each time the program is run.

Well, well! A GO TO loop and a FOR-NEXT loop in the same program! Run the program, and see how it works. Remember, when you see the question mark on the screen, the program is waiting for you to "input" a color code from 1 through 16. If you enter a number that is outside this range, you'll see this error message on the screen:

```
* BAD VALUE IN 20
```

(Remember, also, that color 1 is transparent, and color 4 is the screen color in the Run Mode, so you won't be able to see these bars on screen.)

Experiment now with the color codes, and change the time delay in line 40 if you want to make the bar stay on the screen longer or disappear faster.

Experiment!

Here's a challenge for you! Can you change the program above to make a single small square of color appear on the screen, instead of a bar? (*Hint:* See Chapter 1, pp. 20-22, review using HCHAR or VCHAR to display a single character.)

"Nested" FOR-NEXT Loops

You've just seen that we can use both a FOR-NEXT loop and a GO TO loop in the same program. It's also possible for us to use more than one FOR-NEXT loop – one inside another – in a program. We call these *nested loops.*

As an example, let's experiment a bit with a program very similar to the one you've just completed. But this time, we'll get a little fancier. We'll make the bar "walk" down the screen, so that it appears in a different position each time the color changes. Type these lines:

```
NEW
10 FOR A=1 TO 16
20 CALL CLEAR
30 CALL COLOR(2,A,A)
40 CALL HCHAR(A+5,3,42,28)
50 FOR B=1 TO 300
60 NEXT B
70 NEXT A
80 END
```

Erases old program

Sets limits for A

Uses current value of A to set color codes

Uses current value of A to define "row" position

Sets limits of B for time-delay loop

Executes time-delay loop

Executes "counting" loop that controls how many times the program is run

Notice that one loop is wholly contained within the other loop. That's why these are called "nested" loops: one is nested inside another.

This program gets a lot of mileage out of the variable A. We're using it to control the number of times the program is repeated (a *loop counter*), to define the color codes for foreground and background, *and* to determine the row position of the color bar.

(Before you run the program, remember that color 1 is transparent and color 4 is the Run Mode screen color. You won't be able to see these bars.)

Now run the program. Does the bar appear to move down the screen? What happens if you shorten the time-delay loop? Try changing line 50 to

```
50 FOR B=1 TO 100
```

and run the program again.

Another interesting change would be to make the bar vertical instead of horizontal. We can do this easily by changing line 40. Type and enter this new line:

new row position

```
40 CALL VCHAR(1,A+5,42,24)
```

number of repetitions

new column position

same character code

When you run the program this time, the bar will be vertical and will move across the screen from left to right.

Now let's examine another program with nested FOR-NEXT loops. The following program displays sixty-four of the alphanumeric characters, codes 32 through 95. (See *Appendix B* for a list of the character codes.) Enter these lines:

```
NEW

10 CALL CLEAR

20 LET CHAR=32

30 FOR ROW=7 TO 14

40 FOR COLUMN=13 TO 20

50 CALL HCHAR(ROW,COLUMN,CHAR)

60 CHAR=CHAR+1

70 NEXT COLUMN

80 NEXT ROW

90 END
```

Starting value for variable CHAR (character code)

Beginning and ending values for row number

Beginning and ending values for column number

Increases numeric code for CHAR by 1

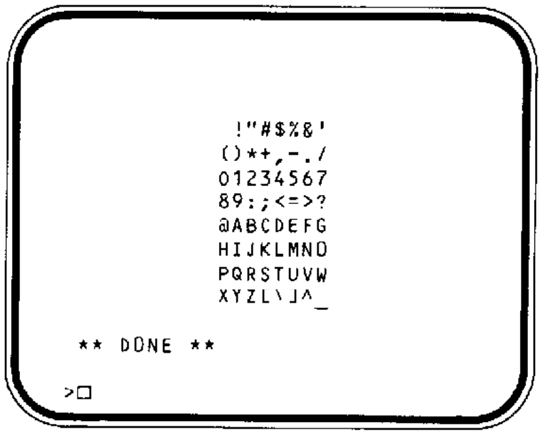
The program will look like this on the screen:

```
 TI BASIC READY

>10 CALL CLEAR
>20 CHAR=32
>30 FOR ROW=7 TO 14
>40 FOR COLUMN=13 TO 20
>50 CALL HCHAR(ROW,COLUMN,CHA
 R)
>60 CHAR=CHAR+1
>70 NEXT COLUMN
>80 NEXT ROW
>90 END
>□
```

There are several things we'd like to point out about this program. First, FOR-NEXT loops *do not* have to start counting at 1. They can begin with whatever numeric value you need to use. Second, the nested loop (FOR COLUMN-NEXT COLUMN) is not just a time-delay loop. It actually controls a part of the program repetition.

Finally, line 50 is called a *wrap-around* line. It has more than 28 characters, so part of it prints on another line on the screen. This is an important point: *program lines can be more than one screen-line long.* In fact, a program line, in general, can be up to four screen lines (112 characters) in length. (The exception is the DATA statement. See the "BASIC Reference" section of the *User's Reference Guide* for an explanation.) Notice that wrap-around lines (that is, the second, third, or fourth screen lines of a program line) are *not* preceded by the small prompting symbol.

Run the program, and the sixty-four characters will be printed in nice, neat rows on the screen:



Hold on! There are only sixty-three characters on the screen! What happened to the other one? Well, there *are* actually sixty-four. Look at the top line, and notice that it appears to be indented one space. That's because *character 32 is a space.* Even though a space doesn't print anything on the screen, it does occupy room on a line, and it *is* a character, as far as the computer is concerned.

Experiment!

Let's add color to the character program above! Enter these lines:

```
22 FOR I=1 TO 8
24 CALL COLOR(I,7,15)
26 NEXT I
```

Try other color combinations until you find your favorite.

Error Conditions with FOR-NEXT

We mentioned earlier that a nested loop *must* be completely contained within another loop. If your program included lines like these,

```
20 FOR A=1 TO 6
30 FOR X=5 TO 10

      ...

80 NEXT A
90 NEXT X
```

Should be nested within the "A" loop.

the computer would stop the program and give you this error message:

```
* CAN'T DO THAT IN 90
```

The computer can't go back inside the completed "A" loop to pick up the beginning of the "X" loop.

Another possible error condition with FOR-NEXT statements is accidentally omitting either the FOR line or the NEXT line. For example, if you attempted to run this program:

```
10 FOR A=1 TO 5
20 PRINT A
30 END
```

the computer would respond with

```
* FOR-NEXT ERROR
```

If you encounter an error message, just list the program (type LIST and press **ENTER**), identify the error, and correct the problem line or lines.

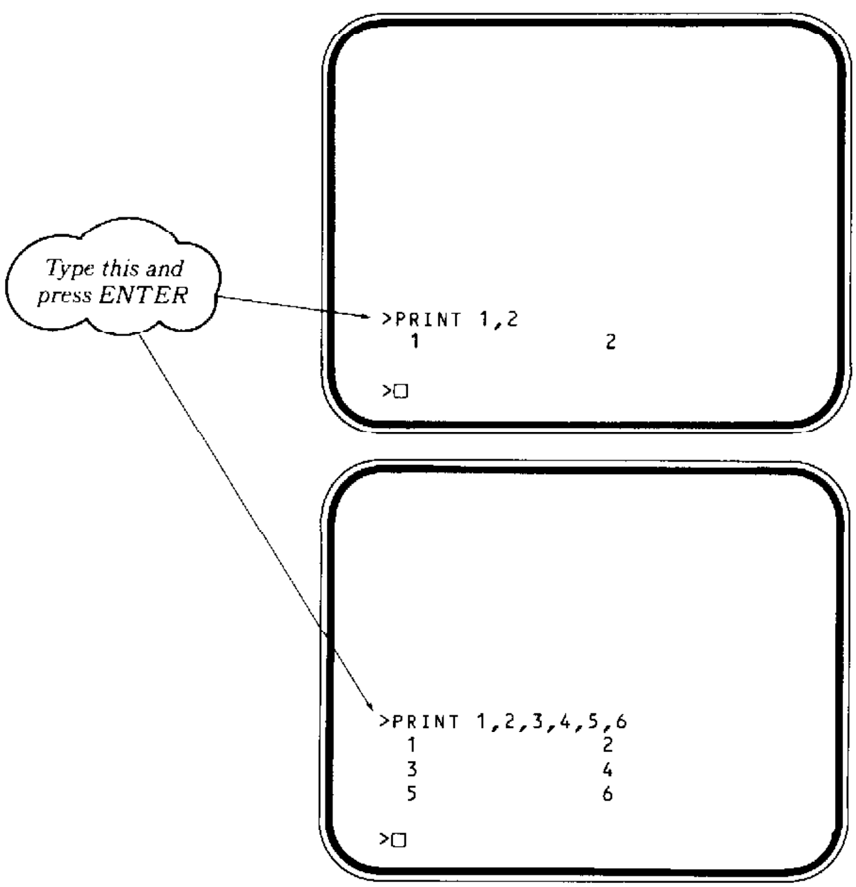
We've given you quite a lot of information now about FOR-NEXT loops, so it's probably time for a change of pace. Let's review a bit of the PRINT material we covered in Chapter 1.

Plain and Fancy PRINTing

While using the PRINT statement in the Immediate Mode, we saw that a difference in spacing occurred when we used a comma or a semicolon to separate numeric values in a PRINT statement. Let's take another look at this.

Spacing with Commas

Try each of the following examples. (In each, we'll assume that the screen has been cleared by typing CALL CLEAR and pressing **ENTER**.)



3

So far we have used only small positive integers. Let's try some simple negative numbers.

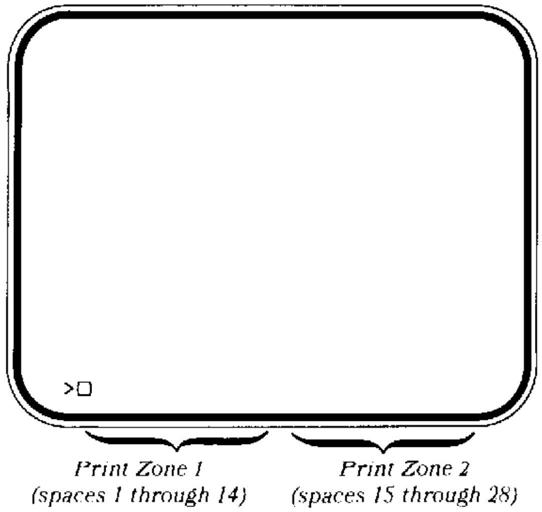
```
>PRINT -1,-2
 -1           -2

>□
```

Now let's try a combination of positive and negative numbers.

```
 PRINT 1,2,-3,-4
 1             2
-3            -4

>□
```

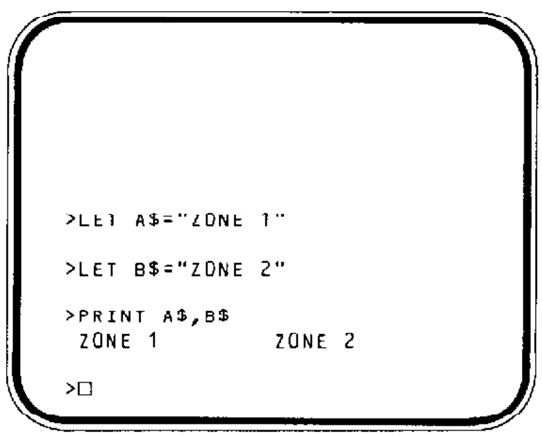
Note that the computer always leaves a space preceding the number for the *sign* of the number. For positive numbers, the plus sign (+) is assumed and is not printed on the screen. For negative numbers, the computer prints a minus sign (−) before the number.

We mentioned in Chapter 1 that there are two *print zones* on the screen line. Each print zone has room for fourteen characters per line.



When you use a comma to separate numeric values or variables in a PRINT statement, the computer is instructed to print only one value in each zone. Therefore, since there are only two print zones on each line, the computer can print a maximum of two values per screen line. If the PRINT statement has more than two items, the computer simply continues on the next screen line until all the items have been printed.

Now let's try some examples with *string variables*, using commas as "separators." (See page 35 of Chapter 2 if you need to review string variables.)



The *strings* (the letters and numbers within the quotation marks) are also printed in different zones on the screen when a comma is used to separate the string variables.

Try this example:

```
>LET A$="ONE"

>LET B$="TWO"

>LET C$="THREE"

>LET D$="FOUR"

>PRINT A$,B$,C$,D$
 ONE            TWO
 THREE          FOUR

>□
```

(Note that, for strings, the computer does not leave a preceding space.)

Spacing with Semicolons

Now let's look at semicolon spacing. Try these examples:

```
>PRINT 1;2
  1  2

>□
```

Aha! The numbers are much closer together.

```
>PRINT 1;2;3
  1  2  3

>□
```

```
>PRINT 1;2;-3;-4;5;-6;7
 1  2 -3 -4  5 -6  7

>□
```

The semicolon instructs the computer *not to leave any spaces* between the values or variables in the PRINT statement. Then why do we see spaces between the numbers on the screen? Two reasons! First, remember that each number is preceded by a space for its sign. Second, every number is followed by a *trailing space.* (The trailing space is there to guarantee a space between all numbers, even negative ones. The way numbers are displayed is discussed in detail in *Appendix D.*)

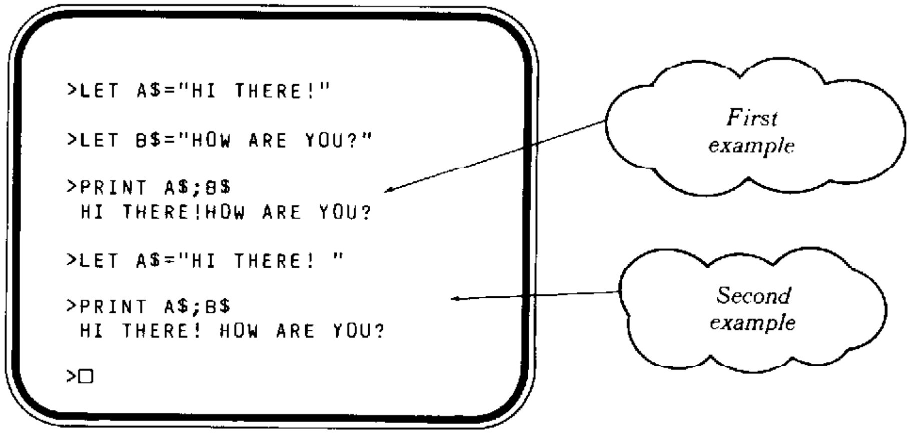
If the semicolon tells the computer to leave no spaces between variables in a PRINT statement, what happens when we use string variables, rather than numeric? Let's try some examples.

```
>LET A$="HI THERE!"

>LET B$="HOW ARE YOU?"

>PRINT A$;B$
 HI THERE!HOW ARE YOU?

>□
```

The two strings are run together. If we want a space to appear between them, then, we must include the space inside one of the sets of quotation marks! For example, let's change A$. Type

```
LET A$="HI THERE! "
PRINT A$;B$
```

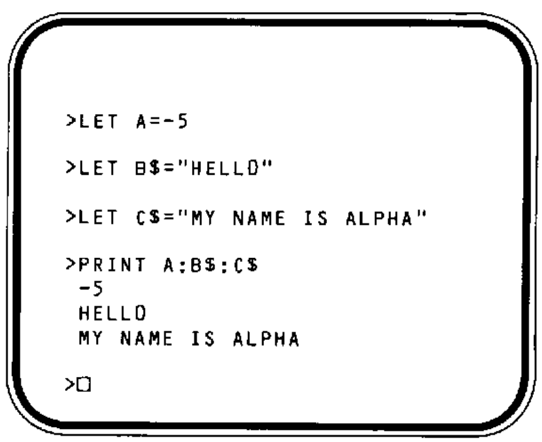
leave one space



Spacing with Colons

There is a third "separator" that can be used: the colon. The colon instructs the computer to print the next item at the beginning of the next line. It works the same way with both numeric and string variables. Enter these lines as an example:

```
LET A=-5

LET B$="HELLO"

LET C$="MY NAME IS ALPHA"

PRINT A:B$:C$
```



To review for a moment, then, these are the three print separators we have used:

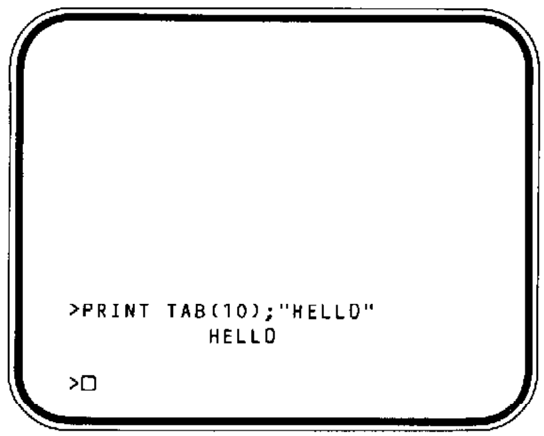
Punctuation mark	*Operation*
Comma	Prints values in different print zones; maximum of two items per line.
Semicolon	Leaves no spaces between items. (The spaces that appear between numbers are results of the built-in display format for numeric quantities.)
Colon	Prints next item on following line.

The TAB Function

Besides these separators there is another method you can use to control the printing on the screen. The TAB function operates very much like a typewriter TAB key:

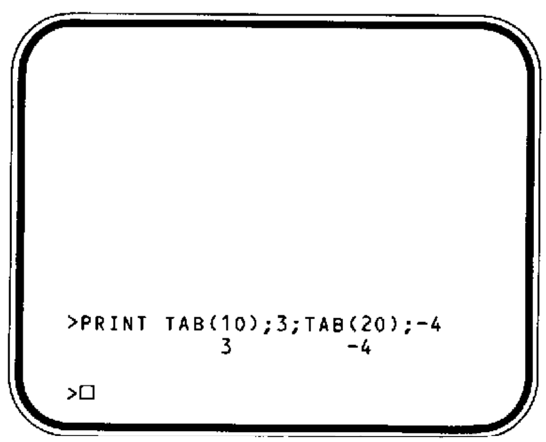
```
PRINT TAB(10);"HELLO"
```

The statement would instruct the computer to begin printing the word HELLO in the tenth column on the screen.



Notice that the "print line" on the screen has 28 columns or character positions (unlike the "graphics line," which has a 32-column "grid"). Thus the first position on the print line counts as column 1. This is where the "P" appears in the word "PRINT" on the previous screen. The last print position on the line is column 28.

You can also use the TAB function more than once in a print statement:



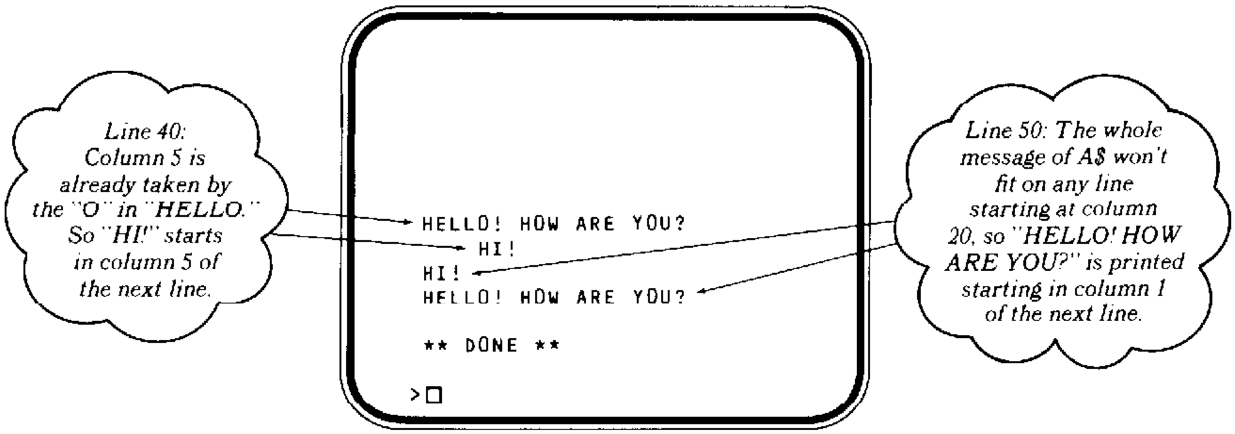
Notice that the first number, 3, is actually printed in column 11, because the preceding or "leading" space (reserved for the sign of the number) occupies column 10, just as the minus sign of the second number occupies column 20.

The TAB function always starts counting in column 1 (the leftmost print position on the line), regardless of where or how many times it appears in the PRINT statement. In the example above, the second number, −4, was printed starting in the twentieth column on the print line, *not* twenty spaces from the position in which the first number, 3, was printed.

What happens, then, if we indicate a column that is already occupied by another message, or if there isn't enough room left on the line to print the message positioned by a TAB? Enter this short program to find the answer:

```
NEW
10 CALL CLEAR
20 LET A$="HELLO! HOW ARE YOU?"
30 LET B$="HI!"
40 PRINT A$;TAB(5);B$
50 PRINT B$;TAB(20);A$
60 END
```

Now run the program:



Notice that separators (semicolons) are also used in the PRINT statement above. Let's try a program to help explore the use of the TAB function and separators. Imagine for a few minutes that you are a loyal football fan, and it's time for the big game of the season. Since you are also a computer fan, you want to program your computer to cheer the team on to victory! So you enter this program:

two *colons!*

```
NEW
10 CALL CLEAR
20 LET A$="GO"
30 PRINT TAB(13);A$::TAB(12);"TEAM"::TAB(13);A$;"!"
40 FOR Z=1 TO 10
50 PRINT
60 NEXT Z
70 FOR Z=1 TO 600
80 NEXT Z
90 GO TO 10
```

Prints an "empty" line.

Note that this is a wrap-around line.

Before you run the program, let's analyze it. Line 10, of course, clears the display screen. Line 20 defines the string variable A$ as GO.

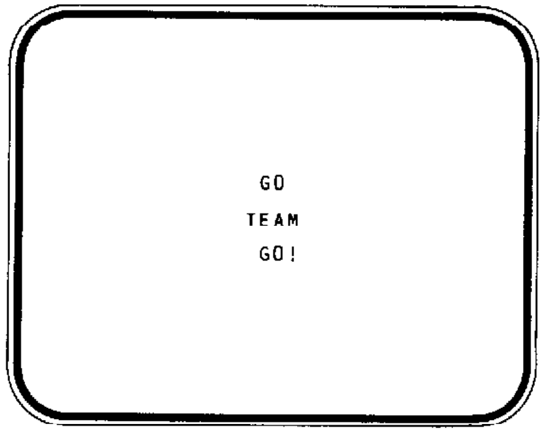
Line 30 is a very, very hard-working line. It might be helpful if we drew a flowchart to describe what's going on here.

`TAB (13)`	Tab over to column 13.
`;`	Do not skip any other spaces.
`A$`	Print GO.
`::`	Print the next print item *two* lines down.
`TAB(12)`	Tab over to column 12.
`;`	Do not skip any other spaces.
`"TEAM"`	Print TEAM.
`::`	Print the next print item *two* lines down.
`TAB(13)`	Tab over to column 13.
`;`	Do not skip any other spaces.
`A$`	Print GO.
`;`	Do not skip any other spaces.
`"!"`	Print an exclamation point.

(You'll have to admit that's a lot of information to pack into one program line, even if it is more than one screen line long!)

The FOR-NEXT loop in lines 40 through 60 will print ten "empty" lines, to position your message in the middle of the screen. Next, lines 70 and 80 form a time-delay loop. Then line 90 instructs the computer to go back to line 10 and start all over again.

Run the program now, and watch your computer cheer!



The words come on at the bottom of the screen, one at a time, and scroll up to the center. Then the screen clears, and the whole process is repeated until you stop the program by pressing **CLEAR.**

By now, your team has probably won the game, and you're ready to try some other messages and formats. Experiment for a while with TAB and the three separators in different PRINT statements before we go on to discuss the arithmetic operations of the computer.

Arithmetic Power

You've been introduced before to the arithmetic powers of your computer, but it's time now to take a more detailed "tour" of some of its mathematical capabilities. For example, what is the answer to this problem:

4+6*5=?

*Remember, * means "multiply" to the computer.*

Let's say, for example, that the answer represents an amount of money you owe a friend. Your friend argues that you owe him $50, because

4+6=10, and
10×5=50.

You, however, don't agree. You say you only owe $34, because

6×5=30
4+30=34

Who is right? Why not ask your computer?

Type `PRINT 4+6*5`
and press **ENTER**.

The answer is 34. How about that! You win!

Order of Operations

There is a commonly accepted order in which arithmetic operations are performed, and your computer performs calculations in that order. In any problem involving addition, subtraction, multiplication, and division, the arithmetic operations will be completed in this way:

> Multiplications and divisions are performed
> before additions and subtractions.

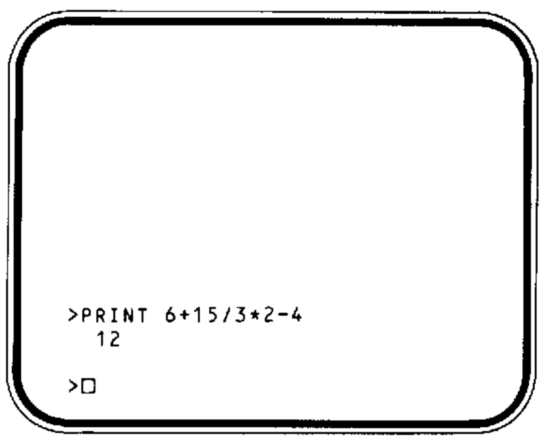
This is the method your computer used to solve the previous example. It first multiplied 6*5 and *then* added the result to 4, giving you a final answer of 34. Now try this example:

```
PRINT 6+15/3*2-4
```

Before you press **ENTER**, let's think about the way the computer will evaluate this problem. Scanning the problem from left to right, the computer will solve it in this order:

15/3=5
5*2=10
6+10=16
16−4=12

Your answer, then, should be 12. Press **ENTER** now, and see the result:



Using Parentheses

Suppose, however, that we want the computer to solve the last problem like this:

(1) Add 6 and 15.
(2) Divide the result by 3.
(3) Multiply that result by 2.
(4) Subtract 4, giving a final result of 10.

We can change the built-in computational order by using *parentheses*. Try this:

```
PRINT (6+15)/3*2-4
```

Press **ENTER**.

The answer, 10, is displayed on the screen, because the computer *has completed the computation inside the parentheses first.* So our new order of operations becomes:

(1) Complete everything inside parentheses.
(2) Complete multiplication and division.
(3) Complete addition and subtraction.

Now try this example:

```
PRINT 8/2*4/2
```

The answer is 8, because

8/2=4
4*4=16
16/2=8

But suppose we entered the problem with parentheses, like this:

```
PRINT 8/(2*4)/2
```

This time, we get a result of .5, because the expression within the parentheses has been solved first:

2*4=8
8/8=1
1/2=.5

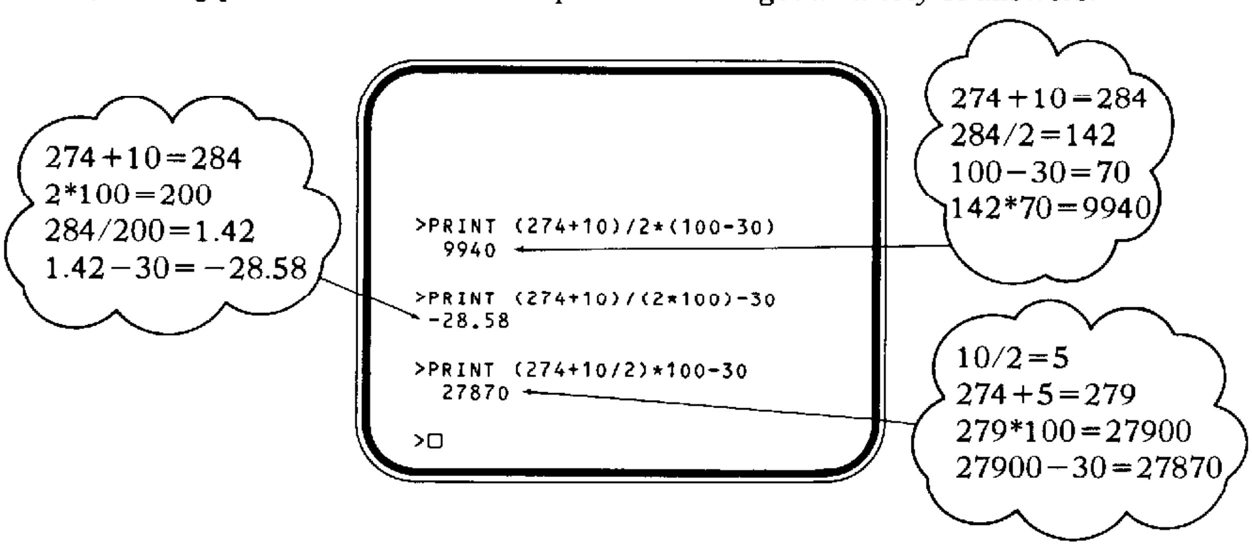
Here's a slightly harder problem to try:

```
PRINT 274+10/2*100-30
```

If we enter the problem just like this, we obtain an answer of 744 because

10/2=5
5×100=500
274+500=774
774−30=744

But by adding parentheses in different places we can get a variety of answers:



Experiment!

Try the following for practice:

38+6−4
38+6−4*2
(38+6−4)*2
((38+6−4)*2)/(6+2)

Note the difference parentheses make.

Rearrange the parentheses in the last problem. How is the answer affected?

Scientific Notation

So far, all the examples we've tried have given results in a normal decimal display form. However, the computer displays very long numbers (more than ten digits) in a special way. Try this program:

```
NEW
10 CALL CLEAR
20 LET A=1000
30 FOR X=1 TO 5
40 PRINT A
50 LET A=A*100
60 NEXT X
70 END
```

When you run the program, the first four results are printed out in the normal form. The last result, however, looks like this:

1.E+11

We call this special form *scientific notation.* It's just the computer's way of handling numbers that won't fit into the normal ten-digit space allotted for numbers.

1.E+11 means 1×10^{11} or 100,000,000,000

As you can see, 1.E+11 represents a very large number!

You'll find a more detailed discussion of the mathematical capabilities and numerical displays of your computer in *Appendix D* (starting on page 127). Be sure to refer to this appendix when you want to explore the computational powers of the computer. For now, however, let's go on to another very useful feature, the INT function.

The INT Function

The INT function gets its name from the word *integer,* meaning a whole number, one that has no fractional part. Integers include zero and all of the positive and negative numbers that do not have any digits after the decimal point.

The best way to learn how the INT function works is by trying it. First, let's work a division problem that doesn't result in a whole number answer. Type

```
PRINT 16/3
```

and press **ENTER**. The answer is 5.333333333.

Now try this example:

```
PRINT INT(16/3)
```

Press **ENTER**.

new answer

```
>PRINT 16/3
 5.333333333

>PRINT INT(16/3)
 5

>□
```

INT kept the whole number part of the answer and threw away the digits after the decimal point! Try another example:

```
PRINT INT(7/6)
```

7/6 = 1.166666666.
INT 7/6 = 1.

The answer is 1; all of the fractional part has been discarded.

How about a real-life problem? Let's say a salesclerk is giving $1.37 in change to a customer. The customer wants as many quarters as possible. How many quarters can be given?

```
PRINT INT(1.37/.25)
```

The answer is 5. Five quarters can be given.

More than one INT function can be used in a PRINT statement. Here's an example:

```
>PRINT INT(1/3);INT(20/9)
 0  2

>□
```

What would happen if you entered these values with the INT function: 8, 8.99, 8.34? Try them and see.

```
>PRINT INT(8)
  8

>PRINT INT(8.99);INT(8.34)
  8  8

>□
```

If you use INT with a whole number (integer), you just get the same number back. In the other two examples, no matter what digits are to the right of the decimal point, the INT function "truncates" or cuts off those digits – that is, it works this way for *positive* numbers. What happens with *negative* numbers?

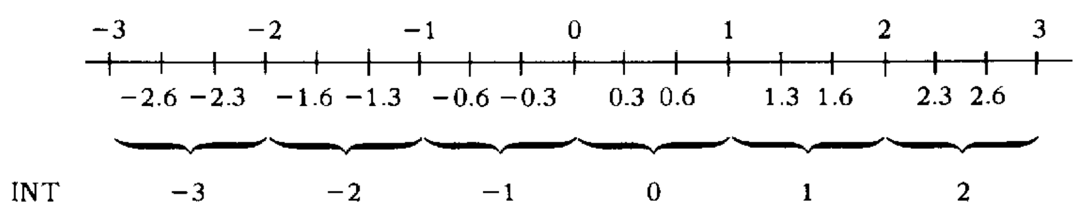
We'll use a program to explore INT and negative numbers. Enter these lines:

```
NEW
10 CALL CLEAR
20 FOR A=1 TO 7
30 PRINT -A/3,INT(-A/3)
40 NEXT A
50 END
```

Now run the program. The screen will show these results:

```
-.333333333       -1
-.666666666       -1
-1                -1
-1.333333333      -2
-1.666666666      -2
-2                -2
-2.333333333      -3
```

So INT(X) – where X represents a number or a mathematical expression – computes the nearest integer *that is less than or equal to X*. Perhaps looking at a *number line* will help to explain.



As you see from the number line, when X has the value −0.3, the nearest integer that is *less than or equal to X* is −1.

One last feature associated with INT is very useful to know. It can appear on the right side of an equals sign in a LET statement. For example, try the next series of lines.

```
>LET A=INT(4/3)+2

>PRINT A
  3

>□
```

In the LET statement, INT(4/3) produces the integer result of 1. This result is added to the constant 2, yielding 3 as a final result. A is then assigned the value of 3 and printed.

Several applications of the INT function are shown in the chapters that follow. For now, try some other experiments with INT so that you become even more familiar with how it works.

Summary of Chapter 3

Chapter 3 has introduced you to some new and powerful TI BASIC capabilities:

FOR-NEXT	You've used this statement to build controlled loops that repeat a part of the program a specified number of times or create a time delay in the program.
PRINT formats	You've learned how to control the spacing of PRINT items using the three separators (comma, semicolon, and colon) and the TAB function.
Computation Order	You've discovered that your computer follows a certain mathematical order in solving problems: 1. Everything in parentheses is computed first. 2. Multiplication and division are done next. 3. Addition and subtraction are performed last.
INT function	You've learned how this function works on both positive and negative numbers that are not *integers* (whole numbers).

These features will help prepare you for the programs that follow in the next chapters.

4 CHAPTER FOUR

Fun and Simulations

In this chapter we'll explore some features of the BASIC language that allow you to create exciting simulations and games.

Many computer programs are *simulations* that imitate some real-world event. With a computer simulation we can imitate an event as simple as the rolling of a single die or as complex as the patterns of animal migration in North America.

As an example of a simulation, we'll enter and run a dice-rolling program in this chapter. Other programs included here explore the games, graphics, and musical capabilities of your computer.

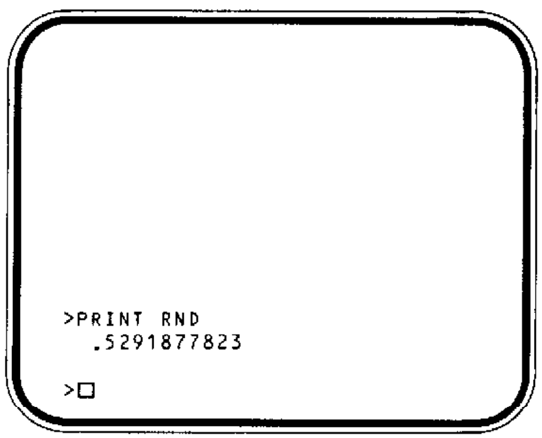
The heart of most games and simulations is the RND function, so let's begin there.

The RND Function

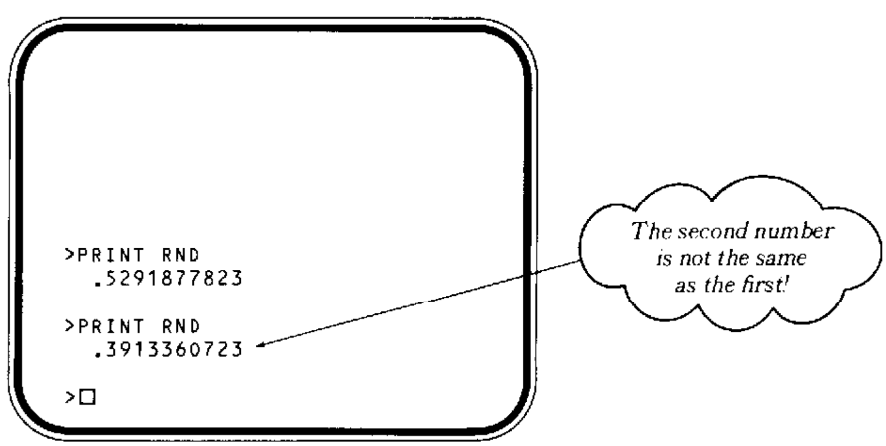
The letters in the name RND are taken from the word RaNDom. To find out what RND does, let's try a few examples in the Immediate Mode.

Clear the screen, and then enter this line:

```
PRINT RND
```



Now try entering the line again:



Here's an interesting situation! Every time we use RND, we get a different number. That's exactly what RND does – it generates *random numbers*.

Now let's try a program that will produce ten random numbers. Enter these lines:

```
20 FOR LOOP=1 TO 10
30 PRINT RND
40 NEXT LOOP
50 END
```

When you've checked your program for errors, run it. A list of ten random numbers will be printed on the screen. Look at the numbers closely. Are any two of the numbers identical?

You may have noticed that all the numbers generated by RND are less than one (1.0) in value. Also, there are no negative numbers. RND is preset to produce only numbers that are greater than or equal to zero and less than one ($0 \leq n < 1$).

Write down the numbers this program produced, and then run the program a second time. Check your written list against the numbers on the screen this time. Very strange! The list of numbers is the same!

This feature of the RND function is important to remember and can be very useful in certain applications. Within a program RND will produce the same sequence of random numbers each time the progran is run.

UNLESS . . . !!

Unless the BASIC statement RANDOMIZE is used in your program.

The RANDOMIZE Statement

Add the RANDOMIZE statement shown below to the program that is still in your computer.

```
10 RANDOMIZE
```

Clear the screen now (type CALL CLEAR; press **ENTER**), and list the changed program on the screen:

```
>LIST
 10 RANDOMIZE
 20 FOR LOOP=1 TO 10
 30 PRINT RND
 40 NEXT LOOP
 50 END
>□
```

Run the program again, and compare the new set of numbers with your written list from the first program run. Are they different this time? They should be!

Experiment!

Continue to experiment with the program until you feel comfortable with RND and RANDOMIZE. For example, try changing line 30 of the previous program to:

```
30 PRINT RND;RND
```

What result does this change have on the program?

If you want the program to generate more or fewer than ten random numbers, just change line 20.

Other Random Number Ranges

The program you just completed generates random numbers between 0 and 1 ($0 \leq n < 1$). Now let's examine ways to increase the range of the numbers we generate.

The RND function can be used as part of any legitimate computation. For example, 10*RND and (10*RND)+7 are both valid uses of RND in TI BASIC. To show what is produced when RND is used in this way, try the following examples:

```
PRINT 10*RND
```

Press **ENTER**

What number appears on the screen? Try the same example again. What number did you get this time?

In both these examples, you should see a decimal point followed by ten digits, or one digit to the left of the decimal point, followed by nine digits to the right of the decimal point. That's because 10*RND produces random numbers in the range of 0 to (but not including) 10, or $0 \leq n < 10$.

Now let's increase the range to this: $0 \leq n < 100$, or random numbers from 0 up to (but not including) 100. Try this:

```
PRINT 100*RND
```

and see what is produced. (Remember, this time you could get *one or two* digits to the left of the decimal point, in the range from 0 through 99.9999 . . .)

Let's use a program to generate some random numbers in the ranges 0 to 10 and 0 to 100. Enter these lines:

```
NEW
10 RANDOMIZE
20 FOR LOOP=1 TO 5
30 PRINT 10*RND,100*RND
40 NEXT LOOP
50 END
```

Every time you run the program, you'll get a different series of numbers.

comma here

Now clear the screen and run the program. Although the numbers you generate on your screen will be different, they'll look something like this:

```
>RUN
 3.196128739    11.32761568
 6.233532821    9.502421843
 7.030941884    33.17351797
 .6689170795    86.40802154
 9.388957913    .7565322811
 ** DONE **

>□
```

Study the differences between the numbers in the left print zone on the screen and those in the right print zone. Can you see that the range is greater in those on the right? Run the program again to produce other numbers.

Suppose we'd like to eliminate all digits to the right of the decimal point and produce random whole numbers (integers). Well, do you remember the INT function we discussed in Chapter 3? This is a job for INT!

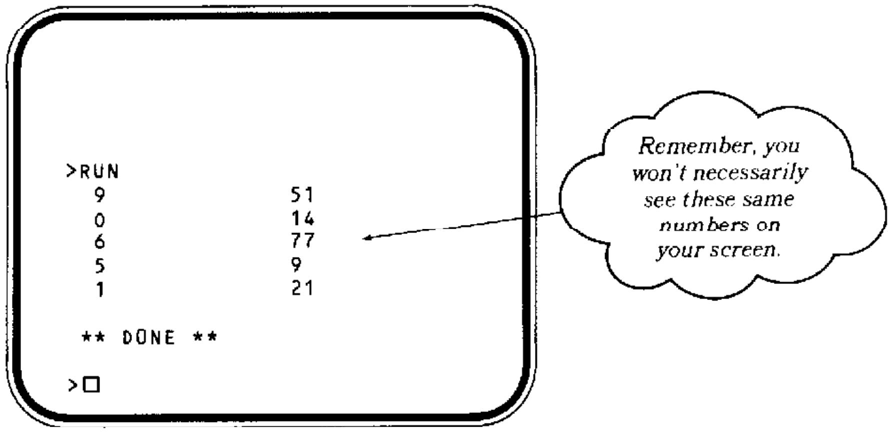
Change the program by typing and entering this new line:

```
30 PRINT INT(10*RND),INT(100*RND)
```

If you list the program now, it will look like this:

```
>LIST
 10 RANDOMIZE
 20 FOR LOOP=1 TO 5
 30 PRINT INT(10*RND),INT(100
*RND)
 40 NEXT LOOP
 50 END
>□
```

When you run the program, the screen will show two series of random whole numbers:



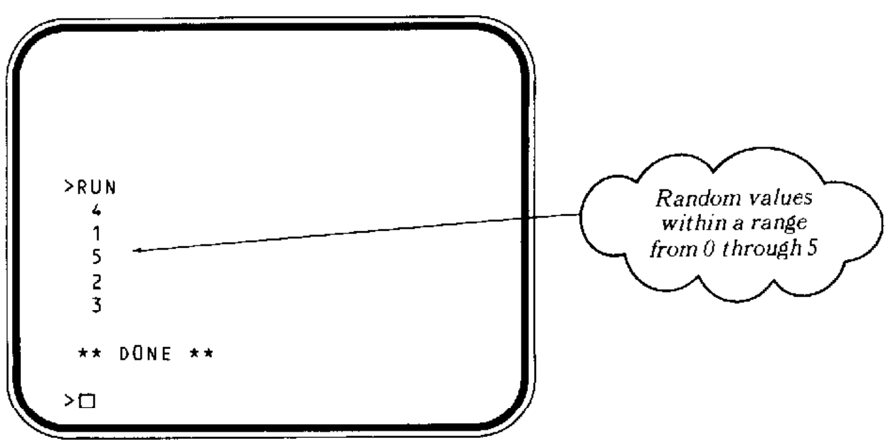
All the numbers on the left side of the screen will have values from 0 through 9, while the numbers on the right have values from 0 through 99. The INT function throws away the digits to the right of the decimal point. The following table summarizes what we have covered so far.

Program Instruction	*Range*
`RND`	0 through .9999 . . .
`10*RND`	0 through 9.9999 . . .
`INT(10*RND)`	0 through 9 (integers only)
`100*RND`	0 through 99.9999 . . .
`INT(100*RND)`	0 through 99 (integers only)

Notice that all these ranges begin with the value of zero. In many games and simulations, however, we need random numbers that start at some other value. For example, to simulate the throw of one die you need a random number generator that produces values from 1 to 6. You have seen that INT(10*RND) gives values from 0 to 9. What would INT(6*RND) produce? Change line 30 in the program to PRINT INT(6*RND) and run the new program.

Type:

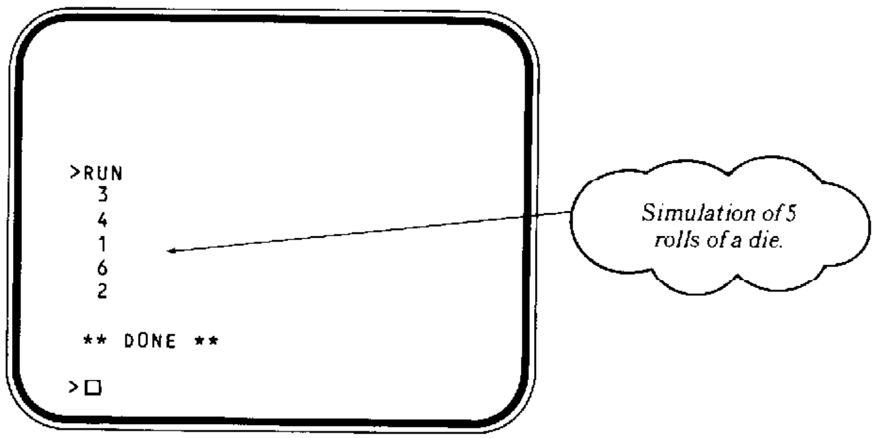
```
30 PRINT INT(6*RND)

CALL CLEAR

RUN
```



Your screen shows a list of five random numbers ranging from 0 to 5. What would happen if we added the value 1 to each item in this list? The resultant numbers would range from 1 to 6. That's just what we need to simulate the throw of a single die. Again, alter the program as shown below and run it.

Type:

```
30 PRINT INT(6*RND)+1

CALL CLEAR

RUN
```



That does it! The program now in your computer is a simulation (imitation) of throwing a single die five times.

A Two-Dice Simulation

At this point we can easily design a program to simulate the throws of two six-sided dice. Before you start, erase the old program by typing NEW. Then enter the following program:

```
5 CALL CLEAR
10 RANDOMIZE
20 INPUT "NUMBER OF ROLLS?":N
30 FOR ROLL=1 TO N
40 DIE1=INT(6*RND)+1
50 DIE2=INT(6*RND)+1
60 PRINT DIE1;DIE2,DIE1+DIE2
70 NEXT ROLL
80 PRINT
90 GOTO 20
```

Input number of rolls to simulate.

Simulate rolls.

Display each die and sum of "spots."

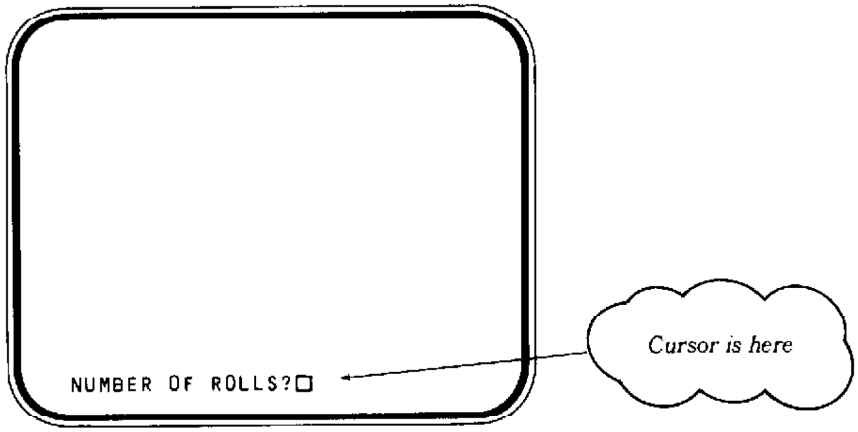
semicolon

comma

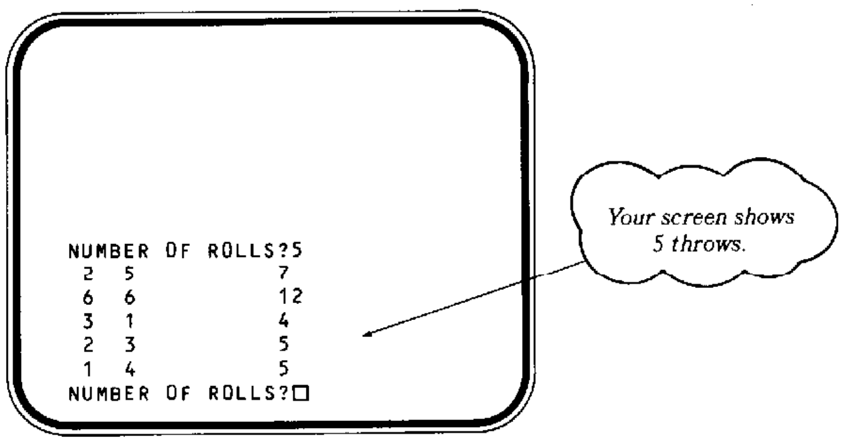
This program prints out the number of "spots" on each die and the sum of the spots on both dice faces. You are asked how many rolls you wish to make at the start of the program. Run the program now and watch what happens.

Type:

```
RUN
```



First, the program prints a request for the number of rolls to make. Enter a number (5, for example) and press the **ENTER** key.



The program keeps looping back to the INPUT request line. (If you want to stop the program, just press **CLEAR**.)

Experiment!

Try entering different values for the number of rolls. What happens if you try 30 rolls? Then make some changes to the program, if you'd like to experiment. For example, how would you alter the program to simulate the throwing of three dice? Two eight-sided dice?

Error Conditions with RND

The error messages produced by an improper usage of RND are essentially the same as the error messages we've mentioned before. Here are some examples:

Typing Errors	*Error Message*
missing operation	
`10 PRINT INT(10RND)`	`*INCORRECT STATEMENT IN 10`
`10 PRINT INT(10*RND`	`*INCORRECT STATEMENT IN 10`
missing close parenthesis	

About the only new error condition we need to mention occurs if you try to use the letters RND as a numeric variable name in a LET or assignment statement. For example, if you type

```
LET RND=5
```

the computer will respond with

```
* INCORRECT STATEMENT
```

This occurs because RND is "reserved" to be used only as a function in TI BASIC. (For a list of all reserved words, see the "BASIC Reference" section of the *User's Reference Guide*.).

Randomized Character Placement

The following program utilizes the INT and RND functions to generate random screen positions for a character you input. First, type NEW and press ENTER to erase your old program; then enter these lines:

```
10 RANDOMIZE
20 INPUT "CHAR CODE?":CODE
30 CALL CLEAR
40 ROW=INT(24*RND)+1
50 COLUMN=INT(32*RND)+1
60 CALL VCHAR(ROW,COLUMN,CODE)
70 GO TO 40
```

Produces a row number from 1 through 24.

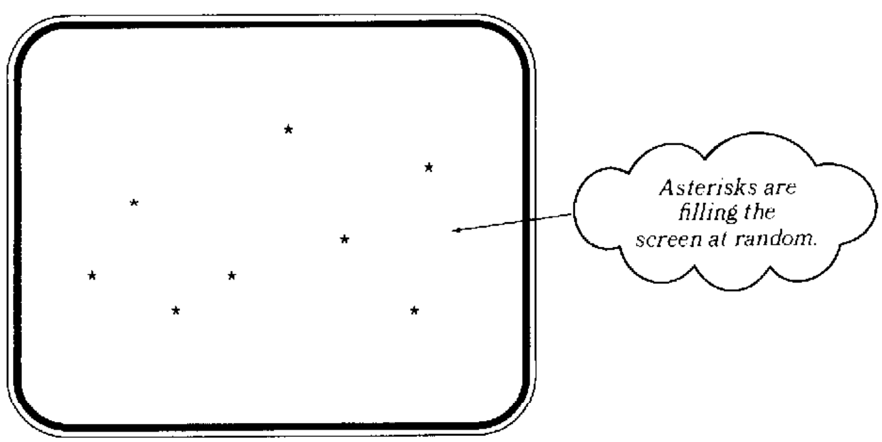
Produces a column number from 1 through 32.

We'll use the character codes 33 through 95; since character 32 is a blank space, we want to avoid entering it when the program asks for a code number.

Before running the program, let's examine a flow chart describing its performance.

Line 10	"Randomizes" the random number series each time the program is run.
Line 20	Stops and asks "CHAR CODE?" Assigns number you enter to the variable CODE.
Line 30	Clears prompting message and input character code from the screen.
Line 40	Produces random integer in range of 0 through 23; adds 1 to value and assigns value to variable ROW.
Line 50	Produces random integer in range of 0 through 31; adds 1 to value and assigns value to variable COLUMN.
Line 60	Prints input character in random position designated by lines 40 and 50.
Line 70	Loops back to produce new random position for character.

Now clear the screen with CALL CLEAR and run the program. For this first example, enter 42 (the character code for the asterisk) as the input for CHAR CODE. The screen will look something like this:



To stop the program just press **CLEAR.** Then try running the program several times, putting in a different character code each time. See if any unusual designs are produced.

When you've finished experimenting with different characters, let's change the program to generate characters at random, as well as placing them randomly on the screen. First we'll have to decide how to set the limits we want for the character range. Here's a general procedure for setting the limits for use with RND:

Subtract the LOWER LIMIT from the UPPER LIMIT.
Add 1.
Multiply that result by RND.
Find the integer (INT) of this result.
Add the LOWER LIMIT.

Now we know that we want 63 characters, with character codes ranging from 33 through 95. So our LOWER LIMIT is 33, and our UPPER LIMIT is 95:

$95-33=62$
$62+1\ =63$

The number we want to multiply by RND is 63, and we must use the INT function:

```
INT(63*RND)
```

Produces random integers from 0 through 62.

Now check the limits established when we add our LOWER LIMIT, 33:

$0+33=33$ (lowest possible character code)
$62+33=95$ (highest possible character code)

INT(63*RND)+33 will give us random whole numbers in the range we need. Type the following new line:

```
20 CODE=INT(63*RND)+33
```

and press **ENTER**. Now clear the screen and list the program to review this change.

```
>LIST
 10 RANDOMIZE
 20 CODE=INT(63*RND)+33
 30 CALL CLEAR
 40 ROW=INT(24*RND)+1
 50 COLUMN=INT(32*RND)+1
 60 CALL VCHAR(ROW,COLUMN,COD
 E)
 70 GO TO 40
>□
```

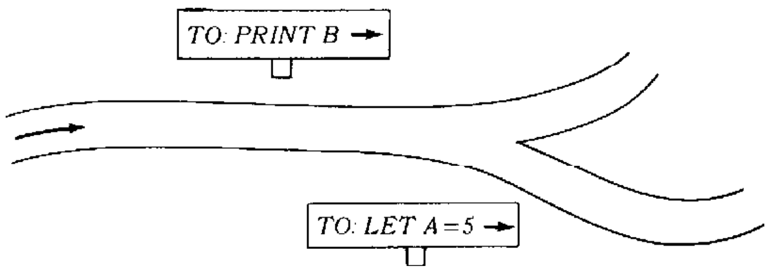
When we run the program this time, the computer will generate a random character code and then print the character in random positions on the screen. (Press **CLEAR** when you want to stop the program.) Run the program several times to see different characters.

Experiment!

By making changes in two lines, you can cause the previous program to print different random characters each time it loops. Try it! (*Hint:* Think about lines 30 and 70.)

The IF-THEN Statement

All the programs we've considered so far in this book have been constructed so that they either run straight through or loop using a GO TO or a FOR-NEXT loop. The IF-THEN statement provides you with the capability of making branches or "forks" in your program. A branch or fork is a point in a program where either one of two paths can be taken, just like a fork in a road.



The general form of an IF-THEN statement looks like this:

IF *condition* THEN *line number*

The *condition* is a mathematical relationship between two BASIC expressions. The *line number* is the program line to which you want the program to branch *if* the condition is true. If the condition is *not true*, then the program line following the IF-THEN statement is executed. For example,

```
30 IF K<10 THEN 70
```

The statement says: *If the value of K is less than 10, then go to line 70 of the program. If K is greater than or equal to 10, then do not branch to line 70. Instead, execute the line following line 30.*

Let's try a demonstration program. Enter these lines:

If new value of K is not less than 10, go on to next line.

```
NEW
10 CALL CLEAR
20 LET K=1
30 PRINT "K-";K
40 LET K=K+1
50 IF K<10 THEN 30
60 PRINT "OUT OF LOOP"
70 END
```

If new value of K is less than 10, go back to line 30 and repeat.

Now run the program.

```
K= 1
K= 2
K= 3
K= 4
K= 5
K= 6
K= 7
K= 8
K= 9
OUT OF LOOP

**DONE**

>□
```

Each time the program reaches line 50, it must make a "true or false" decision. When K is less than 10, the IF condition (K<10) is true, and the program branches to line 30. When K equals 10, however, K<10 is false. The program then executes line 60 and stops.

We mentioned earlier that the condition is a mathematical relationship between two expressions. In the example you've just seen, the mathematical relationship was <, or "less than." There are a total of six relationships that can be used in the IF-THEN statement:

Relationship	*Mathematical Symbol*	*BASIC Symbol*
Equal to	=	=
Less than	<	<
Greater than	>	>
Less than or equal to	$\leq$	<=
Greater than or equal to	$\geq$	>=
Not equal to	$\neq$	<>

Suppose we changed line 50 in the program to this:

```
50 IF K<=10 THEN 30
```

How would the program's performance be affected? Try it! Enter the new line, and then run the program again.

Now, the program prints the value of K all the way through 10, because the new line says, "If K is *less than or equal to 10,* branch to line 30."

Experiment!

The IF-THEN statement can be a powerful tool in program development. Try this program for a graphics application:

```
NEW
10 CALL CLEAR
20 CALL COLOR(2,5,5)
30 LET K=1
40 CALL HCHAR(K,K+1,42)
50 K=K+1
60 IF K<25 THEN 40
70 K=1
80 CALL HCHAR(K,K+3,42)
90 K=K+1
100 IF K<25 THEN 80
110 GOTO 110
```

Can you follow this pattern to create more than two diagonal lines?

Error Conditions with IF-THEN

Like most TI BASIC statements, the IF-THEN statement is pretty particular about its form. The main errors that can occur in using the IF-THEN statement are shown below:

`20 IFA=B THEN 200`	(No space after IF)
`20 IF A=BTHEN 200`	(No space in front of THEN)
`20 IF A=B THEN200`	(No space after THEN)
`20 IF A==B THEN 200`	(Invalid relational symbol combinations)
`20 IF A= THEN 200`	(No expression on one side of the relational symbol)

All of the above conditions produce an error message either when entered or during the running of the program, along with a reference to the line number of the statement in which the error occurs.

If the line number referenced in an IF-THEN statement does not exist, the program stops and produces a message saying that the line number referenced in the statement is not found in the program. For example (using the line above), if 200 is not a valid line number in your program, you see this error message:

```
* BAD LINE NUMBER IN 20
```

Games and Music

The remainder of this chapter explores color graphics and sound through special games applications. Several of the programs are based on a number-guessing game you may have played before. You'll also find that both the RND function and the IF-THEN statement are used extensively in the programs.

4

A Number-Guessing Program

In this game the computer generates a secret number from 1 to 100, using the RND function, and asks you to guess the number. The program tells you if your guesses are larger, smaller, or equal to the secret number. When you guess the number, the program chooses another number and begins the game again.

Type NEW, press **ENTER**, and enter these lines:

Select random number from 1 through 100.

```
10 CALL CLEAR
20 SECRET=INT(100*RND)+1
```

Your guess is input here.

```
30 PRINT "I HAVE A SECRET NUMBER!"
40 PRINT
50 INPUT "WHAT IS YOUR GUESS?":GUESS
```

If you guess correctly, the program branches to line 130.

```
60 IF GUESS=SECRET THEN 130
70 IF GUESS>SECRET THEN 100
```

If your guess is too large, the program branches to line 100.

```
80 PRINT "TOO SMALL!"
90 GOTO 110
100 PRINT "TOO BIG!"
110 PRINT "TRY AGAIN."
120 GOTO 40
130 PRINT "YOU GUESSED IT!"
140 PRINT "LET'S PLAY AGAIN!"
150 FOR DELAY=1 TO 1000
160 NEXT DELAY
```

Delay so you can read the "victory" message.

```
170 GOTO 10
```

Notice that two IF-THEN statements are used in the program, at lines 60 and 70. In line 60, if the guess is *not equal* to the secret number, the condition in the IF-THEN statement is false, and the program proceeds to line 70. If the guess *is equal* to the secret number, the program branches to line 130 and prints the victory message.

At line 70, we test to see if the guess is larger than the secret number. If the guess is larger than the number, the condition is true, and the program branches to line 100. If the guess is smaller than the number, the condition is false, and the program proceeds to line 80.

Now run the program. When it asks "WHAT IS YOUR GUESS?" just type in a number from 1 through 100, and press **ENTER**. Here's an example of what might appear on the screen:

```
I HAVE A SECRET NUMBER!
WHAT IS YOUR GUESS?35
```

First guess

```
TOO SMALL!
TRY AGAIN.
```

Guess is too big

```
WHAT IS YOUR GUESS?75
TOO BIG!
TRY AGAIN.

WHAT IS YOUR GUESS?50
```

Still too big

```
TOO BIG!
TRY AGAIN.
```

Aha.
I must be close!

```
WHAT IS YOUR GUESS?40
TOO SMALL!
TRY AGAIN.

WHAT IS YOUR GUESS?41
```

Is it here? No!

```
TOO SMALL!
TRY AGAIN.

WHAT IS YOUR GUESS?42
```

Try again

```
TOO SMALL!
TRY AGAIN.

WHAT IS YOUR GUESS?45
YOU GUESSED IT!!
LET'S PLAY AGAIN!
```

Did it!!

```
I HAVE A SECRET NUMBER!
WHAT IS YOUR GUESS?
```

The computer will start a new game each time you guess the correct number. When you want to stop playing, just press **CLEAR.**

Notice also that we did *not* include the RANDOMIZE statement. Therefore, the program will generate the same series of random numbers each time you run it! If you want to make the program create a new set of random numbers each time, just add this line:

```
15 RANDOMIZE
```

4

A Tone-Guessing Program

A novel version of the number-guessing program can be created using the sound capabilities of your computer. This program generates a random tone from 131 cycles per second through 247 cycles per second. (If you need to review the CALL SOUND statement and the frequency limits of the computer, see Chapter 1, page 17.) Your job is to guess the frequency of the tone! The program lets you know if your guess is lower, higher, or equal to the frequency of the random tone that is generated. When you guess the correct frequency, the program plays the tone three times and begins the game again.

So type NEW, press **ENTER**, and enter the new program.

```
10 CALL CLEAR
20 TONE=INT(117*RND)+131
30 PRINT "HERE'S THE TONE!"
40 PRINT
50 CALL SOUND(1000,TONE,2)
60 INPUT "GUESS, PLEASE?":GUESS
70 IF GUESS=TONE THEN 160
80 IF GUESS>TONE THEN 110
90 PRINT "TOO LOW!"
100 GOTO 120
110 PRINT "TOO HIGH!"
120 CALL SOUND(1000,GUESS,2)
130 PRINT "TRY AGAIN."
140 PRINT
150 GOTO 30
160 PRINT "YOU GUESSED IT!"
170 FOR PLAY=1 TO 3
180 CALL SOUND(100,TONE,2)
190 NEXT PLAY
200 FOR DELAY=1 TO 500
210 NEXT DELAY
220 GOTO 10
```

Generate a tone between 131 and 247 cycles per second.

Play the tone.

Input your guess – a number from 131 through 147.

Play your guess, if incorrect.

Play your correct guess 3 times.

Line 20 may need a little explanation. If the lowest tone we want is 131 cycles per second and the highest is 247 cycles per second, how do we set our random number limits? Well, INT(117*RND) produces numbers from 0 through 116, and

$0+131=131$ (our desired lower limit)
$116+131=247$ (Our desired upper limit)

Now run the program. The information that appears on the screen is similar to the number-guessing program. The only difference is that in this program your guess is "played" back to you by the computer.

If you'd like to change the tone limits, you can do so easily by changing line 20. For example, suppose you'd rather hear a series of higher tones – perhaps in the range from 262 cycles per second through 392 cycles per second. How would you rewrite line 20 to generate these tones?

Also, you may want to add the RANDOMIZE statement to create a new series of random tones each time you run the program. If so, just enter this new line:

```
15 RANDOMIZE
```

Color Up!

Next, let's examine two color programs. The first program creates ten randomly placed horizontal bars – of a color you input, and of random lengths. Then the program stops for you to input a new color code.

You'll notice that we've used IF-THEN statements in a new way (lines 30 and 40). We test the input color code to be sure it's valid. If it isn't, the program gives you a specially written "error message."

```
10 CALL CLEAR
15 RANDOMIZE
20 INPUT "COLOR PLEASE?":C
25 CALL CLEAR
30 IF C<1 THEN 200
35 IF C>16 THEN 200
40 FOR LOOP=1 TO 10
45 ROW=INT(24*RND)+1
50 REPEAT=INT(28*RND)+1
55 CALL COLOR(2,C,C)
60 CALL HCHAR(ROW,3,42,REPEAT)
65 FOR DELAY=1 TO 100
70 NEXT DELAY
75 NEXT LOOP
80 GO TO 10
200 PRINT "BAD COLOR CODE!"
210 PRINT "MUST BE 1 TO 16."
220 PRINT "TRY AGAIN!"
240 FOR DELAY=1 TO 500
250 NEXT DELAY
260 GO TO 10
```

Test input color code.

Generate a row number from 1 through 24.

Generate a random number of repetitions from 1 through 28.

Print color bar.

Special error message for invalid color code.

When you run the program, you'll see all of the bars begin at column 3, near the left-hand edge of the display. Their lengths, however, are random, as are their horizontal positions on the screen. After ten bars of the input color are placed, the program clears the screen and asks you for a new color code.

Remember to avoid putting in color codes 1 (transparent) and 4 (the screen color in the Run Mode). Although these are valid codes, you won't be able to see the bars.

The next program is a game that contests two colors against each other. A winning color is randomly chosen. The program is the longest you've seen yet, so we'll provide some explanations as we go along. Here's the program:

```
NEW

10 CALL CLEAR
20 INPUT "FIRST COLOR?":C1
30 IF C1<1 THEN 700
40 IF C1>16 THEN 700

50 INPUT "SECOND COLOR?":C2
60 IF C2<1 THEN 800
70 IF C2>16 THEN 800

80 CALL CLEAR

90 COLORTEST-INT(2*RND)+1

100 FOR LOOP=1 TO 50

110 ROW=INT(24*RND)+1
120 COLUMN=INT(32*RND)+1

130 IF COLORTEST=1 THEN 160

140 LET A=C2
150 GOTO 170

160 LET A=C1

170 CALL COLOR(2,A,A)
180 CALL HCHAR(ROW,COLUMN,42)

190 NEXT LOOP

200 FOR DELAY=1 TO 500
210 NEXT DELAY

220 GOTO 10

700 PRINT "BAD COLOR CODE!"
710 PRINT "MUST BE 1 TO 16."
720 PRINT "TRY AGAIN."
730 GOTO 20

800 PRINT "BAD COLOR CODE!"
810 PRINT "MUST BE 1 TO 16."
820 PRINT "TRY AGAIN!"

830 GOTO 50
```

Accept first color code and check for validity. (lines 20–40)

Accept second color code and check for validity. (lines 50–70)

Pick randomly either Color One or Color Two. (line 90)

Set loop to print 50 colored squares on screen. (line 100)

Select random position for color square. (lines 110–120)

If Color One "wins," branch to line 160 and assign value of C1 to A. (line 130)

If Color Two "wins," assign value of C2 to A and go to line 170. (lines 140–150)

Print color square in random position on screen. (line 180)

Go through loop again, until 50 squares are on the screen. (line 190)

Delay to observe pattern; then start new game. (lines 200–220)

Print error message if color codes are invalid. (lines 700–830)

Two people can play against each other, or you can play against yourself by putting in both color codes, just to see which "wins" the game. (Again, avoid entering color codes 1 and 4.)

Random Notes

We've used CALL SOUND earlier in a program that played notes from a musical scale. (See Chapter 2, pages 39-40.) If we modify that program, adding the IF-THEN statement and the RND function, we can make the computer play some interesting (but not necessarily enjoyable) "music." Here's how:

```
NEW
10 LET C=262
15 LET D=294
20 LET E=330
25 LET F=349
30 LET G=392
35 LET A=440
40 LET B=494
45 LET C2=523
50 RANDOMIZE
55 NOTE=INT(8*RND)+1
60 TIME=INT(901*RND)+100
65 VOLUME=2
70 IF NOTE=1 THEN 200
75 IF NOTE=2 THEN 300
80 IF NOTE=3 THEN 400
85 IF NOTE=4 THEN 500
90 IF NOTE=5 THEN 600
100 IF NOTE=6 THEN 700
105 IF NOTE=7 THEN 800
110 NOTE=C2
115 CALL SOUND(TIME,NOTE,VOLUME)
120 GOTO 55
200 NOTE=C
210 GOTO 115
300 NOTE=D
310 GOTO 115
400 NOTE=E
410 GOTO 115
500 NOTE=F
510 GOTO 115
600 NOTE=G
610 GOTO 115
700 NOTE=A
710 GOTO 115
800 NOTE=B
810 GOTO 115
```

Set up a musical scale from "middle C" through "high C."

Randomly select 1 of the 8 notes.

Randomly select a duration from 100 through 1000 milliseconds.

Check which note to play. Notice that we don't have to check for the last note — "high C" (C2). It will automatically be selected if none of the first 7 notes are selected.

Play the note.

Define NOTE.

Now run the program and enjoy the "music." When you're ready to "stop the music," just press **CLEAR.**

You might like to experiment with this program in various ways. For example, do you notice anything different in the "music" if you change lines 60 and 65 to

```
60 TIME=500
65 VOLUME=5
```

A Musical Interlude

Now that we've let the computer play its "music," let's play some music of our own! With this program we can use the keyboard to input the notes we want to play. Enter these lines:

```
NEW
10 CALL CLEAR
15 LET C=262
20 LET D=294
25 LET E=330
30 LET F=349
35 LET G=392
40 LET A=440
45 LET B=494
50 INPUT "NOTE ":A$
55 IF A$="C" THEN 100
60 IF A$="D" THEN 200
65 IF A$="E" THEN 300
70 IF A$="F" THEN 400
75 IF A$="G" THEN 500
80 IF A$="A" THEN 600
85 IF A$="B" THEN 700
90 GOTO 50
100 NOTE=C
110 GOTO 800
200 NOTE=D
210 GOTO 800
300 NOTE=E
310 GOTO 800
400 NOTE=F
410 GOTO 800
500 NOTE=G
510 GOTO 800
600 NOTE=A
610 GOTO 800
700 NOTE=B
800 CALL SOUND(100,NOTE,2)
810 GOTO 50
```

This time, we'll only define 7 notes.

leave one space

Accept "note."

Check for the letter key pressed on the keyboard.

Not A-G! Do it again.

Play "note."

Return for new note.

When you run the program, the program will ask you for a note. You then press one of the letter keys (A,B,C,D,E,F, or G), followed by the **ENTER** key. For example, when the screen shows

```
NOTE?
```

Cursor blinks until you enter a "note."

and you press these keys:

A (ENTER)

the "note" A will play. The screen keeps a record of the keys you depress:

```
NOTE C

NOTE D

NOTE E

NOTE F
```

Having to press the **ENTER** key for each note slows down your musical performance a bit, doesn't it? What can we do about this problem?

The CALL KEY Routine

There is a routine that permits the transfer of one character from the keyboard directly into a program. The routine is CALL KEY. If you alter the current program in the following way, you don't have to press the **ENTER** key after hitting the key for each note.

Enter:

must be zero

Character code of depressed key ends up here.

```
50 CALL KEY(0,NOTE,STATUS)
55 IF STATUS=-1 THEN 50
60 IF STATUS=0 THEN 50
65 IF NOTE=67 THEN 100
70 IF NOTE=68 THEN 200
75 IF NOTE=69 THEN 300
80 IF NOTE=70 THEN 400
85 IF NOTE=71 THEN 500
90 IF NOTE=65 THEN 600
95 IF NOTE=66 THEN 700
```

Status indicator. If:
1 = new key since last time
-1 = same key as last time
0 – no key depressed

Check character code of depressed key to see if it is A, B, . . ., G.

Here's how CALL KEY works. Each character on the keyboard has a numeric code. When a key is depressed, the character code of that key is assigned to the second variable in the KEY routine. In this example, the character code is assigned to the variable NOTE. The last variable in the KEY routine is a status indicator. The indicator lets the program know what has occurred on the keyboard. If you keep holding down the same key, the STATUS is minus one. If you press a key different from your last entry, the STATUS is one (1). If you don't press any key, the STATUS is zero (0). When you run the program, nothing appears on the screen as you press the keys. The program simply plays the note you request. So go ahead – make a little music!

The CALL KEY routine allows you to create "your own kind of music," and the routine can also be used in many games and simulations where single-character input values are requested. The CALL KEY routine speeds up the input of data by eliminating the need to press the **ENTER** key after your data entry.

Summary of Chapter 4

This chapter has given you an idea of the many interesting games and simulations you can develop with your computer. You've discovered these new features:

RND	Allows you to generate random numbers.
RANDOMIZE	Insures that each series of random numbers generated by a program will be different.
IF-THEN	Provides conditional branching capabilities in a program.
CALL KEY	Permits the transfer of a keyboard character directly into a program, without pressing **ENTER**.

Congratulations! You've accomplished a lot of computer programming!

The following chapter deals only with computer graphics. You'll learn how to define your own characters and how to make "animated" patterns on the screen. Just turn the page for some more exciting experiences!

5 CHAPTER FIVE

Computer Graphics

This chapter continues developing programs that demonstrate the graphics capabilities of your computer. The programs deal with the use of color, animation and the generation of your own graphics characters on the screen.

The intent in this chapter is to give you some hints and examples that will help you expand your enjoyment and use of your computer. As you begin to develop your own graphics applications, you may want to refer to this material for ideas on how to approach the use of color and graphics. In time, you'll discover other ways to create specific programs and simulations. But for now the programs that are given here will demonstrate several techniques you'll enjoy using.

Blocks of Color

In previous chapters we've experimented with several programs that placed color lines or squares on the screen. The program below shows you how to create larger blocks of color.

```
NEW
10 INPUT "COLOR CODE?":C
20 IF C<1 THEN 10
30 IF C>16 THEN 10
40 CALL CLEAR
50 CALL COLOR(2,C,C)
60 FOR I=1 TO 4
70 CALL VCHAR(2,I+2,42,4)
80 CALL VCHAR(19,I+2,42,4)
90 CALL VCHAR(2,I+24,42,4)
100 CALL VCHAR(19,I+24,42,4)
110 CALL VCHAR(12,I+13,42,4)
120 NEXT I
130 INPUT "PRESS ENTER KEY":
KEY$
140 GOTO 10
```

Check for valid color code.

Set color.

Each CALL VCHAR places a vertical strip of color on the screen.

Wait here.

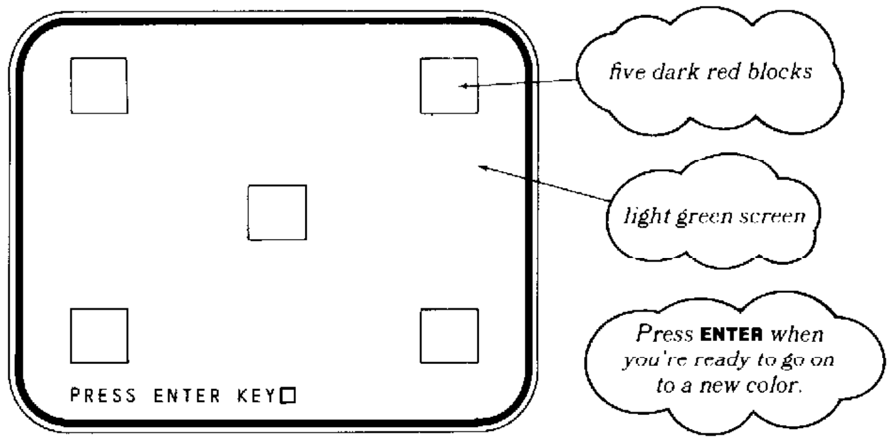
First the program stops and asks you to input a color code. (See *Appendix* C for the list of colors that correspond to the valid color codes, 1 through 16.) When you enter a code, the screen clears, the blocks of color are displayed, and the program waits for you to press a key before continuing. Notice the special use of the INPUT statement in line 130. We are just using INPUT to stop the program until we are ready to go on.

Now clear the screen and run the program. First you'll see

```
COLOR CODE?□
```

flashing cursor

Remember that color code 1 is transparent and code 4 is the screen color in the Run Mode. So let's enter 7 (dark red) as our first color code. When you press **ENTER**, you'll see this:



Experiment with several colors. Find the ones that produce clear sharp images against the normal color of the screen. Can you see how this technique could be used to create checkerboard patterns or the board area for a game like tic-tac-toe?

(Each block in this program is four "characters" wide and four "characters" high. If you'd like to see the "characters," press **CLEAR** to stop the program.)

The CALL SCREEN Statement

So far, the color of the screen in the Run Mode has always been light green. Suppose, however, that we would prefer a different color as a background for our color design. Easy! All we have to do is to add a simple statement that changes the Run Mode screen color:

```
CALL SCREEN(11)
```

Color code for screen color we want, enclosed in parentheses.

Let's edit the color block program we just entered so that we can use a different screen color as a background for our blocks. Enter these lines:

```
32 INPUT "SCREEN COLOR?":A
33 IF A<1 THEN 32
34 IF A>16 THEN 32
45 CALL SCREEN(A)
```

Input screen color code.

Check for valid code

Change screen color from light green to color you input.

Now run the program again. This time, you'll be asked to enter *two* color codes. The first code determines the color of the blocks; the second sets the color of the screen.

5

Experiment!

Experiment with different color combinations. Which give you the sharpest, clearest design? Which are most pleasing in an artistic sense? Then try changing the block design produced by the program. For example, can you make the blocks rectangular?

Patterns

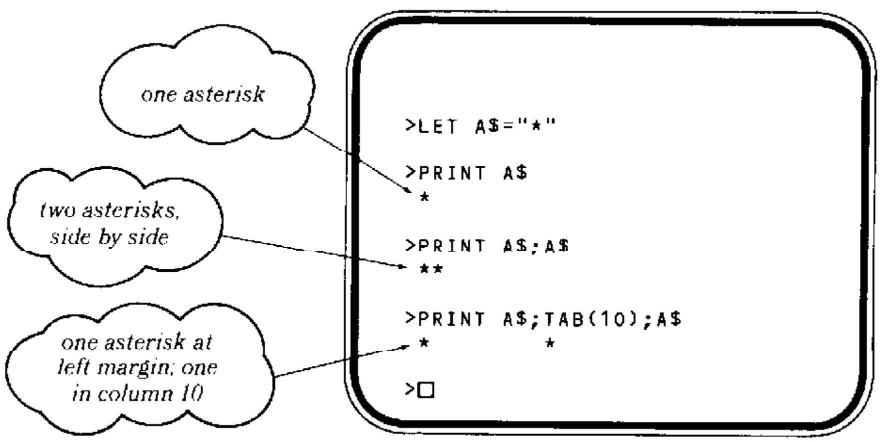
The two programs we'll develop next continue our exploration of computer graphics by showing how to construct patterns out of standard characters. The statements and functions used in the program are elements of BASIC you already know; however, you may see some new applications of these features.

Rectangles and Squares

The first program allows you to place a rectangle or square of standard characters on the screen. Instead of using CALL HCHAR or CALL VCHAR and identifying the character by its *character code*, we'll assign a character to a string variable from the keyboard.

Try these examples in the Immediate Mode:

```
LET A$="*"
PRINT A$
PRINT A$;A$
PRINT A$;TAB(10);A$
```



Try a few more Immediate Mode experiments on your own. For example, what would happen if you redefine A$ as "***" or as "()"? Try it and see what results! (If you need to review the TAB function and the print separators, see Chapter 3, page 57.)

This method is convenient if you want to print only a short line of characters. But what if you want to print a long line or vary the line length or character the program prints? INPUT statements and a FOR-NEXT loop will solve the problem. Type NEW; then enter this program:

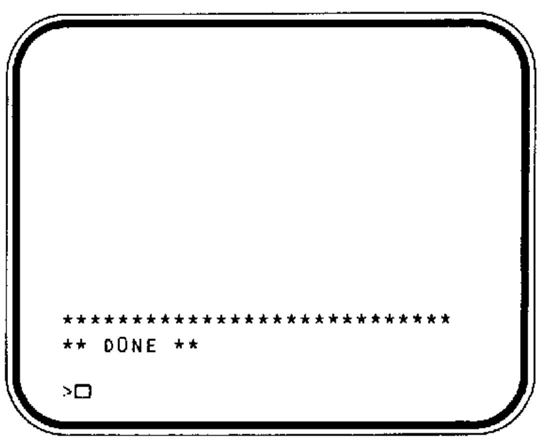
```
20 INPUT "CHARACTER?":A$
40 INPUT "WIDTH?":W
60 CALL CLEAR
80 FOR X=1 TO W
100 PRINT A$;
120 NEXT X
140 END
```

Accept character.

Accept number of characters to be printed.

semicolon

When you run the program, you'll first be asked to input the character you want to use. Just type the character and press **ENTER**. Then you'll be asked for the "width" or the number of characters in the line you want to print. Type in the number and press **ENTER** to continue the program. Let's say that you entered * as the character and 28 as the width. The screen will look something like this:



(Note that the semicolon in line 100 causes the characters to be printed in an unbroken row.)

Run the program a few times, entering different characters and lengths. Then let's try adding some program lines that will allow us to make rectangles and squares of characters.

Enter these new lines:

```
40 INPUT "SIZE(WIDTH,HEIGHT)":W,H
70 FOR Y=1 TO H
130 PRINT
135 NEXT Y
140 GOTO 40
```

Replaces old line 40.

Replaces old line 140.

There are a couple of items that need to be explained about these lines. First, notice in line 40 that we are using one INPUT statement to assign values to *two* variables! When you input the width and height, you'll need to use this form:

comma

The number of characters you want in each row.

```
8,5
```

The number of rows you want.

Second, lines 70 and 135 set up a loop on the variable Y. Your original "X loop" is now nested inside the "Y loop."

Finally, line 130 prints an "empty" line. This line is needed to clear away the semicolon (;) in line 100 so that a new row will begin the next time the program loops through the "Y loop." (As you've seen already, the semicolon causes the characters to be printed on the same line throughout the loop on X.)

Before we list the program to see the changes, let's add a few more lines. We can use IF-THEN statements to "build in" some tests:

If character is XX, stop the program.

```
25 IF A$="XX" THEN 150
45 IF W+H=0 THEN 20
150 END
```

If both width and height are 0, ask for new character.

Here's what these tests provide. Line 25 gives you a handy way to stop the program by pressing the **X** key twice and then pressing **ENTER** when you're asked for a character input. If you want to experiment with a different character, all you have to do is to enter **0,0** as size inputs. The test in line 45 then sends you back to line 20 to input a new character.

Now clear the screen and list the program:

```
LIST
20 INPUT "CHARACTER?":A$
25 IF A$="XX" THEN 150
40 INPUT "SIZE(WIDTH,HEIGHT)
":W,H
45 IF W+H=0 THEN 20
60 CALL CLEAR
70 FOR Y=1 TO H
80 FOR X=1 TO W
100 PRINT A$;
120 NEXT X
130 PRINT
135 NEXT Y
140 GOTO 40
150 END
```

If A$ = XX, stop!

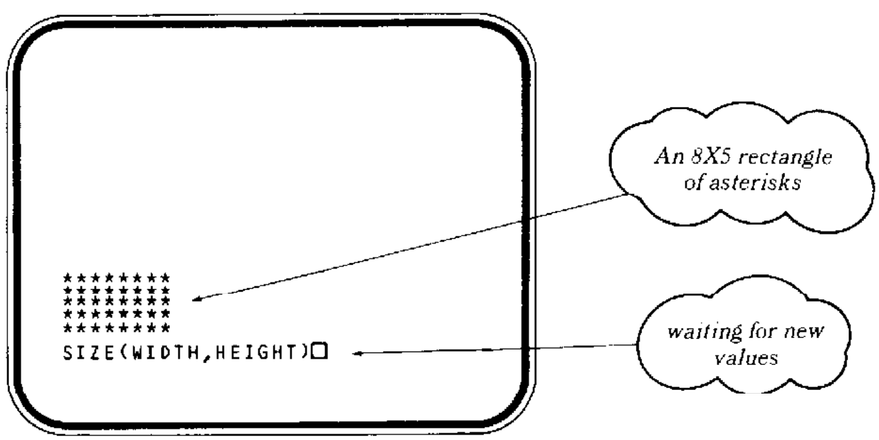
If W and H both = 0, get new character input.

Print a line of characters.

Start a new line.

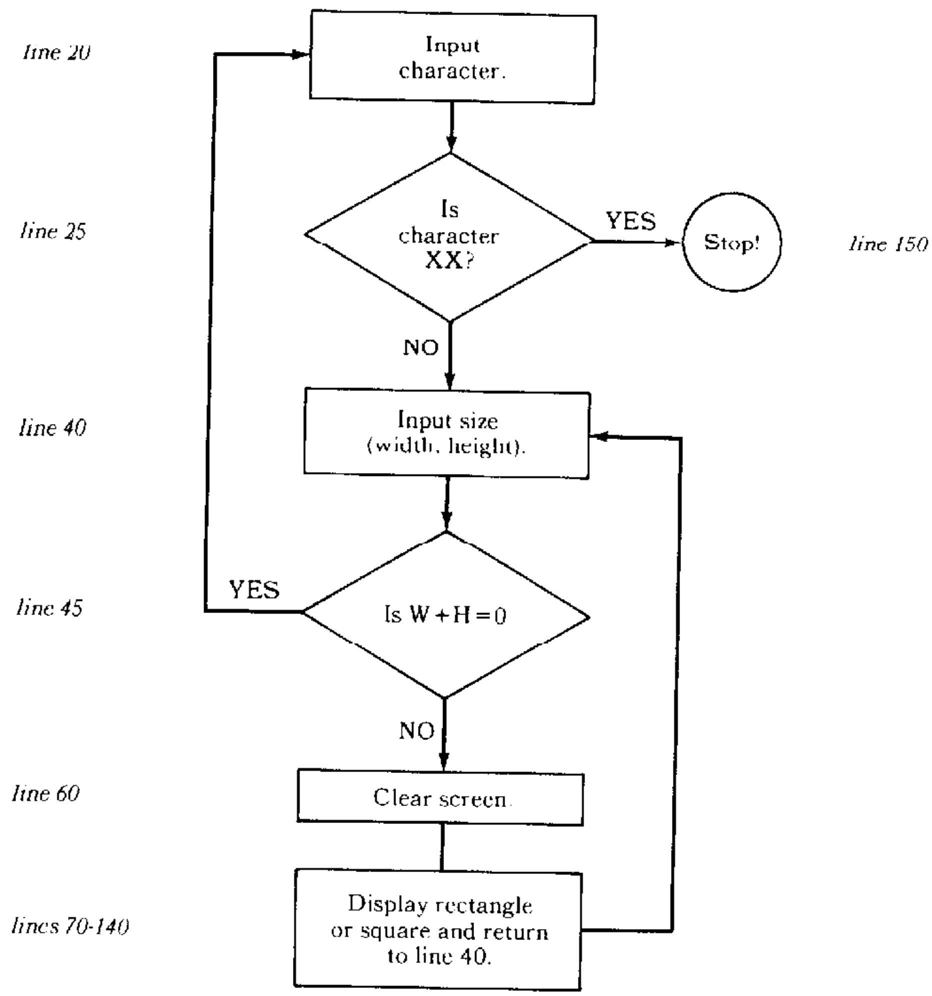
Return for new size inputs.

Clear the screen again and run the program. For this example, enter * when the program asks CHARACTER? Then enter **8,5** when you're asked for width and height:



Next, enter the same value for both width and height, such as 8,8 or 5,5. With these inputs the program will create a square, rather than a rectangle.

Perhaps a flowchart will help to describe how the program works. The following diagram doesn't show the whole program in detail; it covers only the parts that relate to program control by input values.



Experiment!

Experiment with the program. Try entering the control values (character input=XX, width and height both = zero) to see how the program reacts. Vary the width and height so that the display fills the screen or makes only tall thin bars and wide flat strips. What happens if you enter a width greater than 28 or a height greater than 24? (Try it and see what happens.) Can you add color to this program?

"Holes"

Let's expand the Rectangles and Squares program one more time. These new lines will create rectangles or squares with a random sprinkling of "holes" (blank spaces) in the display field. Enter the following lines:

```
15 RANDOMIZE
85 IF INT(2*RND)=0 THEN 100
90 PRINT " ";
95 GOTO 120
```

*INT(2*RND) produces a 0 or a 1.*

One space, enclosed in quotation marks and followed by a semicolon.

Now clear the screen and list the changed program, so that we can discuss the effect of these additions.

```
LIST
15 RANDOMIZE
20 INPUT "CHARACTER?":A$
25 IF A$="XX" THEN 150
40 INPUT "SIZE(WIDTH,HEIGHT)
":W,H
45 IF W+H=0 THEN 20
60 CALL CLEAR
70 FOR Y=1 TO H
80 FOR X=1 TO W
85 IF INT(2*RND)=0 THEN 100
90 PRINT " ";
95 GOTO 120
100 PRINT A$;
120 NEXT X
130 PRINT
135 NEXT Y
140 GOTO 40
150 END
```

If true, go to line 100 and PRINT the character. If false PRINT a blank space (line 90).

Skip line 100.

The test with the RND function in line 85 causes a character to be printed whenever INT(2*RND) is equal to 0, a space when INT(2*RND) is equal to 1. Thus, approximately half of the time the program prints a character and half of the time a space. Run the program now, and observe the kind of pattern that emerges.

Experiment!

You'll be able to see the "holes" better by making the character into a color block. Add these lines to the program:

```
10 INPUT "COLOR?":C
45 IF W+H=0 THEN 10
50 CALL COLOR(3,C,C)
```

Now, when the program asks, COLOR?, type and enter a color code from 1 to 16. Notice that, to see the character in color, you'll have to enter a character input from set #3 (0,1,2,3,4,5,6, and 7 are the characters in this set).

5

Animation

Animation is the illusion of movement. In order to achieve this illusion in your graphics programs, it's necessary to keep changing your character or sets of characters. The following programs demonstrate some of the techniques used to create flashing and moving graphics on the screen.

Flashing Letters

One way to create a flashing graphic is to print a character (or set of characters), delay the program, clear the screen, delay the program again, and then repeat the process. The clearing of the screen and the delays have the effect of turning the character "on and off," making it appear to flash. Let's try a program that flashes the letter A in the center of the screen.

```
NEW
10 CALL VCHAR(12,16,65)
20 FOR DELAY=1 TO 200
30 NEXT DELAY
40 CALL CLEAR
50 FOR DELAY=1 TO 100
60 NEXT DELAY
70 GOTO 10
```

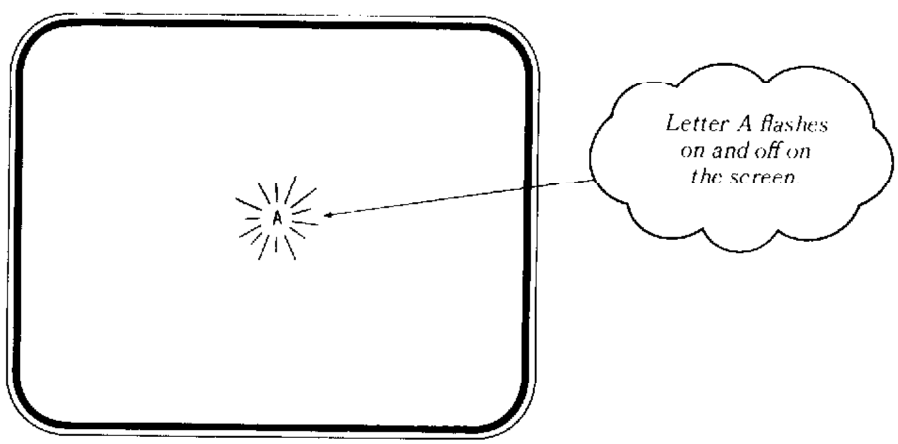
Numeric code for A

Pause after printing.

Delay after clearing.

Repeat.

Now clear the screen and run the program.



Another way to simulate flashing is to replace one character with another in the same spot on the screen. Let's revise our program so that it alternately flashes A and B. We can do this easily by entering a new line 40:

```
40 CALL VCHAR(12,16,66)
```

same position

code for B

Since we're replacing A with B, we don't have to clear the screen between printing the characters. However, we may want to add a CALL CLEAR at the beginning of the program. So enter this line:

```
5 CALL CLEAR
```

and run the revised program. Do A and B appear to flash alternately on the screen? (You may want to increase the time delay in line 50, so that A and B will each stay on the screen the same length of time.)

From flashing characters to flashing color squares is an easy step, so we'll examine next a program that places a flashing color square on the screen.

Flashing Color Squares

With this program we want to create a color square that flashes on the screen. We'll write the program so that we can input the color we want, and we'll use character 42 (the asterisk, in character set 2) to make our square. Here's the program:

```
NEW
10 CALL CLEAR
20 INPUT "COLOR CODE?":X
30 CALL CLEAR
40 CALL COLOR(2,X,X)
50 CALL VCHAR(12,16,42)
60 FOR DELAY=1 TO 200
70 NEXT DELAY
80 CALL CLEAR
90 FOR DELAY=1 TO 200
100 NEXT DELAY
110 GOTO 40
```

Accept color code (1-16).

Define color for character set 2, which contains the asterisk, and make foreground and background the same color.

Print square of color.

Clear screen

Repeat.

Now run the program. First, it asks

```
COLOR CODE?
```

and waits for you to input a valid color code. The codes are 1 through 16; remember, however, that code 1 is transparent and code 4 is the normal screen color in the RUN Mode. Squares of these colors will not show up on the screen.

When you type in a color code and press **ENTER**, you'll see the square flashing near the center of the screen.

Next, let's change the program to create *two* color squares that alternately flash on the screen. To do so, we'll need to input two color codes. So enter these lines first:

```
20 INPUT "COLOR1?":X
25 INPUT "COLOR2?":Y
```

Accept color for first square.

Accept color for second square.

Now we'll replace our original line 80 with two new lines, to set the color and display the second square:

```
80 CALL COLOR(2,Y,Y)
85 CALL VCHAR(12,16,42)
```

Set color for second square.

Print second square.

Let's review these changes by listing the program. Clear the screen; then type LIST and press **ENTER**:

```
LIST
10 CALL CLEAR
20 INPUT "COLOR1?":X
25 INPUT "COLOR2?":Y
30 CALL CLEAR
40 CALL COLOR(2,X,X)
50 CALL VCHAR(12,16,42)
60 FOR DELAY=1 TO 200
70 NEXT DELAY
80 CALL COLOR(2,Y,Y)
85 CALL VCHAR(12,16,42)
90 FOR DELAY=1 TO 200
100 NEXT DELAY
110 GOTO 40
```

First color square is on the screen.

Second color square is on the screen.

Repeat.

Select your two colors and run the program, typing in the color codes as the program asks for them. The two color squares will alternately flash on the screen.

Experiment with several color combinations to find those that give a good contrast. Here are a few examples to try:

Color 1	*Color 2*
6	5
11	14
14	16
9	11

Moving Color Squares

With just a few simple changes in the previous program, we can make the color squares move across the screen as they flash. Add these lines:

```
35 FOR I=3 TO 28
50 CALL VCHAR(12,I,42)
85 CALL VCHAR(12,I,42)
105 CALL CLEAR
110 NEXT I
120 GOTO 10
```

Now list the program to review the changes:

```
CALL CLEAR
LIST
10 CALL CLEAR
20 INPUT "COLOR1?":X
25 INPUT "COLOR2?":Y
30 CALL CLEAR
35 FOR I=3 TO 28
40 CALL COLOR(2,X,X)
50 CALL VCHAR(12,I,42)
60 FOR DELAY=1 TO 200
70 NEXT DELAY
80 CALL COLOR(2,Y,Y)
85 CALL VCHAR(12,I,42)
90 FOR DELAY=1 TO 200
100 NEXT DELAY
105 CALL CLEAR
110 NEXT I
120 GOTO 10
```

When you've checked the program for accuracy, run it. Starting at column 3, the squares flash and travel across the screen, ending at column 28. Then the screen clears, and the program asks you for new color inputs.

Experiment!

If you want to speed up the flashing, shorten the time delay loops in lines 60 and 90. For a challenge, you might like to make the program flash *three* color squares! How would you do it?

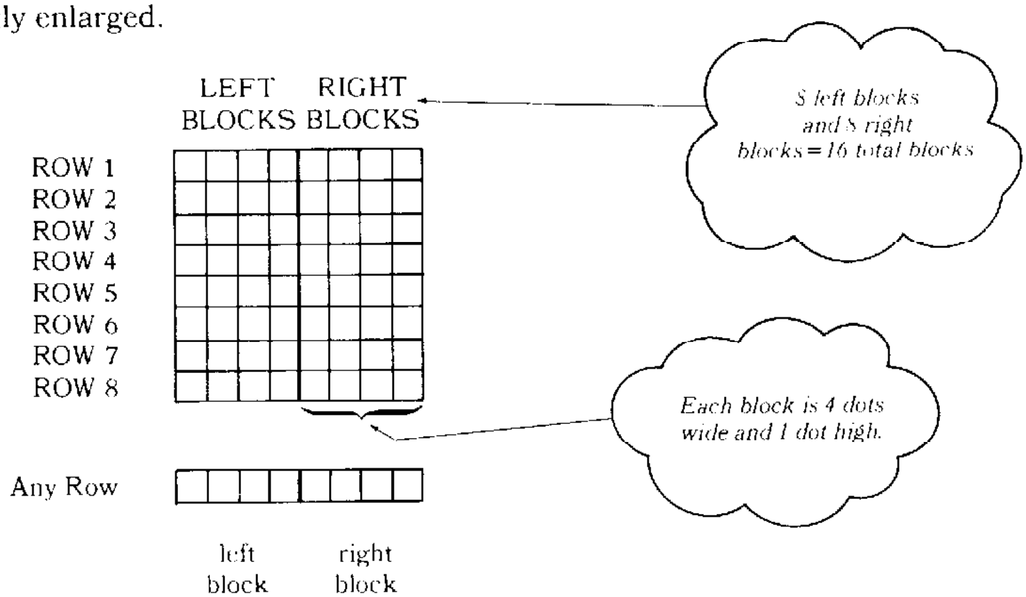
By this time you've seen several examples of the kind of graphics you can create with the standard characters of your computer. Next we'll show you how to develop characters of your own.

5

The CALL CHAR Statement

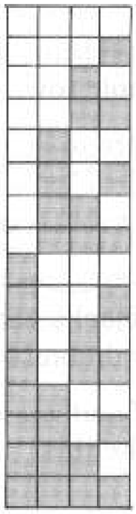
The CALL CHAR statement gives you the capability of creating your own screen characters. In our first program we'll *redefine* some of the standard characters. Before we redefine a character, however, we must first look at the way a character is represented on the screen.

Each printing position on the screen is made up of sixty-four tiny dots. The dots are arranged in eight rows of eight dots each. Each row is partitioned into two blocks of four dots each. The diagrams below show how an 8-by-8 grid of dots would look if it were greatly enlarged.



A character on the screen, either a standard character or one that you invent, is formed by dots within the 8-by-8 grid. By turning some dots "on" and leaving others "off," a character is created. Leaving all the dots "off" creates the space character (character code 32), for example. Turning all the dots "on" produces a solid spot on the screen.

All the standard characters are automatically set so that they turn on the appropriate dots to produce the images you have seen. To create a new character, we must tell the computer which dots to turn on or leave off in each of the 16 blocks within the printing region that contains the character. In your computer a shorthand system is used to specify which dots are on or off within a particular block. The table that follows contains all the possible on/off conditions for the dots within a given block and the shorthand notation for each condition.



BLOCKS	*DOT CODE* *(0=off, 1=on)*	*SHORT-HAND CODE*
	0000	0
	0001	1
	0010	2
	0011	3
	0100	4
	0101	5
	0110	6
	0111	7
	1000	8
	1001	9
	1010	A
	1011	B
	1100	C
	1101	D
	1110	E
	1111	F

Let's take a look at one row (two blocks) to see how the "shorthand code" works.

	LEFT BLOCK				RIGHT BLOCK			
Any row→								
Dot Code	0	1	0	1	1	0	1	1
Shorthand	5				B			

The shorthand code for the row, then, is 5B.

The shorthand codes for an entire grid can be determined block by block, just by converting the on/off conditions of each row. The following example provides a translation of an entire grid into the shorthand code.

	LEFT BLOCK				RIGHT BLOCK				CODE
ROW 1		X	X	X	X	X	X		7E
ROW 2	X		X			X		X	A5
ROW 3	X							X	01
ROW 4	X			X	X			X	99
ROW 5	X							X	81
ROW 6	X		X	X	X	X		X	BD
ROW 7	X							X	81
ROW 8		X	X	X	X	X	X		7E

left block code

right block code

7EA5819981BD817E

shorthand for all blocks

Therefore, if we want to "define" a character shaped as the X's on the grid indicate, we enter all the shorthand codes of the blocks as a single "string":

```
"7EA5819981BD817E"
```

In the shorthand code, then, one number or letter represents a whole block (4 dots) on the grid. Two letters and/or numbers represent a whole row.

Based on the table, if all the dots in all the blocks were to be turned on, the shorthand code for this condition would be:

```
"FFFFFFFFFFFFFFFF"
```

One F for each block

This code may seem long, since it represents all 16 blocks within the grid. But it is still shorter than trying to write down all 64 separate conditions dot by dot.

Once you've decided which dots you want on and off and worked out the code, you're ready to use the CALL CHAR statement. It looks like this:

```
CALL CHAR(33,"FFFFFFFFFFFFFFFF")
```

code for character you are redefining

"string" that turns the dots on and off

comma

Let's try a simple program that redefines character code 33(!) as a character with all the dots turned on. The new character is then printed in the center of the screen, giving you a chance to see exactly how big one of the individual print areas really is. Enter these lines:

```
NEW
10 CALL CLEAR
20 CALL CHAR(33,"FFFFFFFFFFFFFFFF")
30 CALL VCHAR(12,16,33)
40 GOTO 40
```

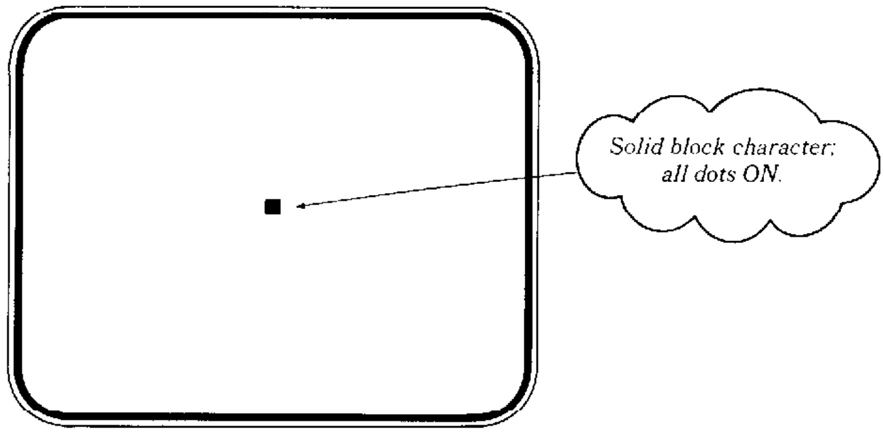
Redefining this character

String of shorthand codes

What does this do?

New character 33 in center of screen

Run the program and observe your newly defined character on the screen!

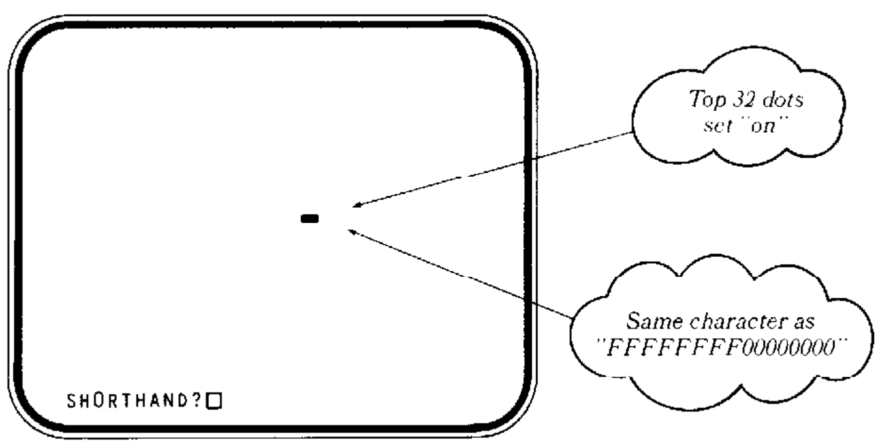


So that you can experiment with other shorthand codes, let's edit the program. Type these new lines:

```
5 INPUT "SHORTHAND?":A$
20 CALL CHAR(33,A$)
40 GOTO 5
```

This time, when you run the program, you'll be asked to input the shorthand code for the character you are defining. Try the following examples.

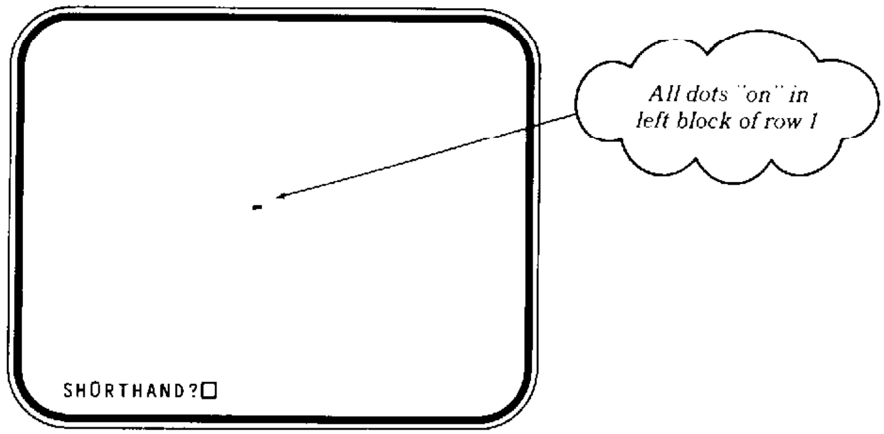
Enter: **FFFFFFFF**



When you stop the program by pressing **CLEAR,** the character that you created changes back into the character from the standard character set. In this case, character code 33 is restored to an exclamation point (!), and that symbol appears near the center of the screen.

Entering FFFFFFFF is the same as entering FFFFFFFF00000000. That is, the CHAR routine fills out the right side of the string variable with zeros when there are less than 16 characters in the string. Knowing this fact allows you to easily examine all the shorthand codes individually. Just enter 0, 1, and so on up to F at the INPUT request.

Enter: **F**



Try different combinations of the shorthand codes. See if you can generate any interesting characters. Then let's revise the program again to print more than just one of our redefined characters. Enter these lines:

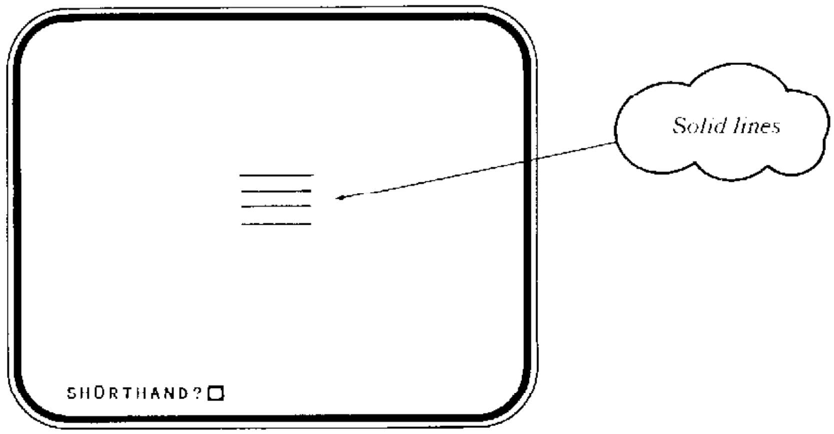
```
30 FOR I=1 TO 4
40 CALL VCHAR(12,I+13,33,4)
50 NEXT I
60 GOTO 5
```

Now list the program to see the changes:

```
LIST
5 INPUT "SHORTHAND?":A$
10 CALL CLEAR
20 CALL CHAR(33,A$)
30 FOR I=1 TO 4
40 CALL VCHAR(12,I+13,33,4)
50 NEXT I
60 GOTO 5
```

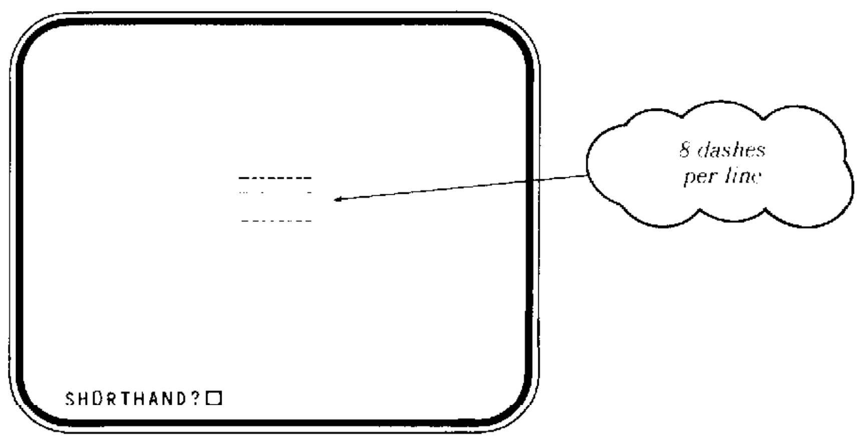
When you run this program and enter a shorthand code for a new character, that character is displayed 16 times in the center of the screen. The 16 characters appear in a square four characters wide by four characters high. Try the following:

Enter: **FF**



A single print of the character with the shorthand code FF puts something like a long dash on the screen. Printing four of these characters side by side draws a line on the screen! To get dashes across the screen you must leave a space by setting two dots in each block "off." To do this, the code is 33.

Enter: **33**



Notice that, when you stop the program, the center of the screen fills with 16 exclamation points (!).

Now enter some other codes and experiment with the program until you feel comfortable with the shorthand codes. To help you work out the codes, draw up several 8-by-8 grids and mark off your "dots-on, dots-off" design. Then figure out the code you need for each block of the grid.

5

A Block Figure with CALL CHAR

Now that you've had some experience with defining your own characters, let's see if we can create a small "human" figure by turning dots on and off.

To begin, you need to create the figure on a character grid worksheet, like the one below. (Later, when you are creating your own characters, you may want to make copies of the worksheet, not only to design your symbols, but also to use in translating the symbol into the shorthand code of the CALL CHAR statement.)

CHAR Worksheet

	LEFT BLOCK	RIGHT BLOCK	CODE	SHORT-HAND CODE	DOTS
ROW 1			___	0	0000
ROW 2			___	1	0001
ROW 3			___	2	0010
ROW 4			___	3	0011
ROW 5			___	4	0100
ROW 6			___	5	0101
ROW 7			___	6	0110
ROW 8			___	7	0111
				8	1000
				9	1001
				A	1010
				B	1011
				C	1100
				D	1101
				E	1110
INPUT TO CHAR: ________________				F	1111

Using the worksheet, we'll mark ones (1's) in the positions where the dots will be turned on:

CHAR Worksheet

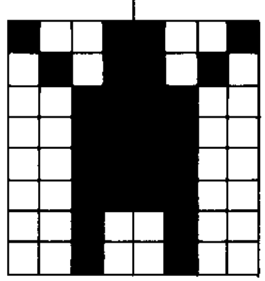
	LEFT BLOCK				RIGHT BLOCK				CODE	SHORT-HAND CODE	DOTS
ROW 1	1			1	1			1	—	0	0000
ROW 2		1		1	1		1		—	1	0001
ROW 3			1	1	1	1			—	2	0010
ROW 4			1	1	1	1				3	0011
ROW 5			1	1	1	1			—	4	0100
ROW 6			1	1	1	1			—	5	0101
ROW 7			1			1			—	6	0110
ROW 8			1			1			—	7	0111
										8	1000
										9	1001
										A	1010
										B	1011
										C	1100
										D	1101
										E	1110
										F	1111

INPUT TO CHAR: ____________________

Now, let's look at the same figure with the "on" dots shaded in, and let's fill in the shorthand codes for developing the character. This form of the worksheet shows you what the character will look like on the screen.

CHAR Worksheet

LEFT BLOCK | RIGHT BLOCK



	CODE	SHORT-HAND CODE	DOTS
ROW 1	99	0	0000
ROW 2	5A	1	0001
ROW 3	3C	2	0010
ROW 4	3C	3	0011
ROW 5	3C	4	0100
ROW 6	3C	5	0101
ROW 7	24	6	0110
ROW 8	24	7	0111
		8	1000
		9	1001
		A	1010
		B	1011
		C	1100
		D	1101
		E	1110
		F	1111

INPUT TO CHAR: 995A3C3C3C3C2424

By filling in the worksheet for both the character and the shorthand codes, we know that one line of our program will be

```
LET A$="995A3C3C3C3C2424"
```

But before we actually start our program, we need to discuss a bit further the process of defining a character. In our previous examples we *redefined* an already existing character, the exclamation point (character code 33). There are other character codes, however, that are *undefined* by the computer. These are available for you to use in building a customized character set in your graphics programs. The undefined character codes are grouped into the following sets (for color graphics):

Set #13	*Set #14*	*Set #15*	*Set #16*
128	136	144	152
129	137	145	153
130	138	146	154
131	139	147	155
132	140	148	156
133	141	149	157
134	142	150	158
135	143	151	159

These extra character codes allow you to design special graphics characters for use in your own programs without giving up the standard keyboard characters. For example, you might want to design differently colored underline characters to highlight certain parts of a displayed message or develop a gameboard on the screen with directions displayed in standard text characters.

These codes and their corresponding set numbers are used in the CALL CHAR, CALL HCHAR, CALL VCHAR, and CALL COLOR statements exactly as we used the defined character codes and their set numbers. Let's use code 128 in our sample program.

OK, we're ready to begin our program. Enter these lines:

```
NEW
10 CALL CLEAR
20 LET A$="995A3C3C3C3C2424"
30 CALL CHAR(128,A$)
40 CALL COLOR(13,2,16)
50 CALL VCHAR(12,16,128)
60 GOTO 60
```

The shorthand code for our "figure."

Define character code 128.

set number

black

white

Display character.

Now run the program and observe the small "person" on the screen. Remember, the figure is only one character in size, so look closely. When you're ready to stop the program, press **CLEAR.**

Would it be possible to animate our little figure? Yes, it would! By changing our program and incorporating one of the techniques we covered under ANIMATION, we can turn our character into Mr. Bojangles, the dancing man!

Mr. Bojangles

As it's written presently, our program defines only one character. To make Mr. Bojangles appear to move, we'll need to define two characters that are alternately displayed in the same position. So we'll go to our CHAR Worksheets to design our two new characters.

CHAR Worksheet

First Figure

	LEFT BLOCK				RIGHT BLOCK				CODE
ROW 1	1			1	1			1	99
ROW 2		1		1	1		1		5A
ROW 3			1	1	1	1			3C
ROW 4			1	1	1	1			3C
ROW 5			1	1	1	1			3C
ROW 6			1	1	1	1			3C
ROW 7		1				1			44
ROW 8	1					1			84

SHORT-HAND CODE	DOTS
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

INPUT TO CHAR: "995A3C3C3C3C4484"

CHAR Worksheet

Second Figure

	LEFT BLOCK				RIGHT BLOCK				CODE	SHORT-HAND CODE	DOTS
ROW 1				1	1				18	0	0000
ROW 2	1			1	1			1	99	1	0001
ROW 3	1	1	1	1	1	1	1	1	FF	2	0010
ROW 4			1	1	1	1			3C	3	0011
ROW 5			1	1	1	1			3C	4	0100
ROW 6			1	1	1	1			3C	5	0101
ROW 7			1				1		22	6	0110
ROW 8			1					1	21	7	0111
										8	1000
										9	1001
										A	1010
										B	1011
										C	1100
										D	1101
										E	1110
										F	1111

INPUT TO CHAR: "1899FF3C3C3C2221"

Now we're ready to edit the program. Enter these lines:

```
20 A$="995A3C3C3C3C4484"
25 B$="1899FF3C3C3C2221"
35 CALL CHAR(129,B$)
60 FOR DELAY=1 TO 100
70 NEXT DELAY
80 CALL VCHAR(12,16,129)
90 FOR DELAY=1 TO 100
100 NEXT DELAY
110 GOTO 50
```

first character

second character

Define character code 129 as B$.

Display first figure.

Display second figure.

Return and display first character, repeating the whole procedure.

Clear the screen and list the changed program so that you can see how it fits together:

```
LIST
10 CALL CLEAR
20 A$="995A3C3C3C3C4484"
25 B$="1899FF3C3C3C2221"
30 CALL CHAR(128,A$)
35 CALL CHAR(129,B$)
40 CALL COLOR(13,2,16)
50 CALL VCHAR(12,16,128)
60 FOR DELAY=1 TO 100
70 NEXT DELAY
80 CALL VCHAR(12,16,129)
90 FOR DELAY=1 TO 100
100 NEXT DELAY
110 GOTO 50
```

Now run the program and watch Mr. Bojangles dance! (To stop the program, press **CLEAR**.)

Experiment!

After running the program a few times, you might like to add a FOR-NEXT loop to make Mr. Bojangles dance across the screen (see page 107 for an example of this technique). Also, try creating other pairs of characters and placing their shorthand codes in lines 20 and 25. Can you turn Mr. Bojangles into an acrobat who flips from his hands to his feet and back again?

As we've mentioned, Mr. Bojangles is pretty small – only one character in size. Not all the designs you can create are limited to this small size. You can combine several small characters to construct bigger graphics that cover more of the screen. Our next program shows how to design a larger graphic using one small color character as our "building block."

The Giant

If you define one special character where all the dots are "on," you can then use it to paint in the rest of a large figure. The following program takes the small character just mentioned and creates a "giant" figure similar to the Mr. Bojangles character. Enter the program and see what it does:

Shorthand code to turn on all the dots in the character.

Set character color — dark blue (foreground and background).

```
NEW
10 CALL CLEAR
20 A$="FFFFFFFFFFFFFFFF"
30 CALL CHAR(128,A$)
40 CALL COLOR(13,5,5)
50 CALL VCHAR(7,15,128,8)
60 CALL VCHAR(7,16,128,8)
70 CALL VCHAR(9,14,128,10)
80 CALL VCHAR(9,17,128,10)
90 CALL VCHAR(7,12,128,3)
100 CALL VCHAR(9,13,128)
110 CALL VCHAR(9,19,128,3)
120 CALL VCHAR(9,18,128)
130 GOTO 130
```

"Build" head and center of torso.

Left torso and leg.

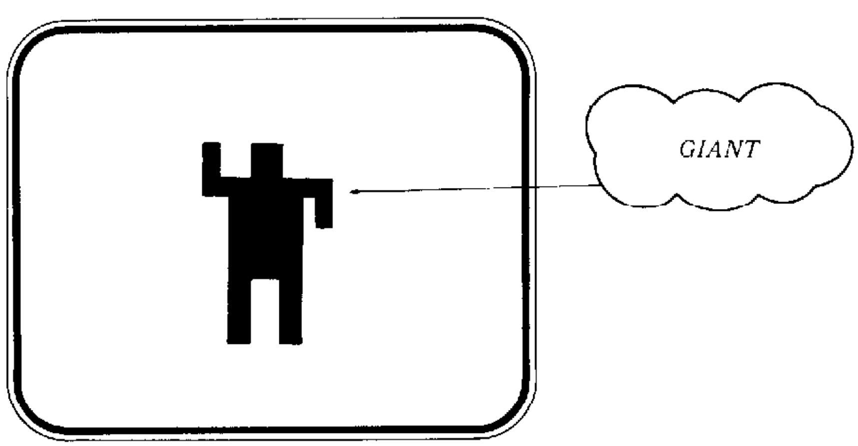
Right torso and leg.

Left arm.

Right arm.

Hold graphic on screen.

When you run the program, you'll see a larger version of Mr. Bojangles:



Our dark-blue "giant" is rather angular and blocky, since it's created from a single angular character. You might like to rework the program, adding extra defined characters that allow you to soften the edges of the figure.

Experiment!

Experiment with some other block designs using your "all-dots-on" character. Then try defining other characters to include in your graphics programs. The examples shown below will help to get you started.

CHAR Worksheet

	LEFT BLOCK				RIGHT BLOCK				CODE
ROW 1								1	01
ROW 2							1	1	03
ROW 3						1	1	1	07
ROW 4					1	1	1	1	0F
ROW 5				1	1	1	1	1	1F
ROW 6			1	1	1	1	1	1	3F
ROW 7		1	1	1	1	1	1	1	7F
ROW 8	1	1	1	1	1	1	1	1	FF

INPUT TO CHAR: "0103070F1F3F7FFF"

CHAR Worksheet

	LEFT BLOCK				RIGHT BLOCK				CODE
ROW 1				1	1				18
ROW 2			1	1	1	1			3C
ROW 3		1	1	1	1	1	1		7E
ROW 4	1	1	1	1	1	1	1	1	FF
ROW 5	1	1	1	1	1	1	1	1	FF
ROW 6		1	1	1	1	1	1		7E
ROW 7			1	1	1	1			3C
ROW 8				1	1				18

INPUT TO CHAR: "183C7EFFFF7E3C18"

CHAR Worksheet

	LEFT BLOCK				RIGHT BLOCK				CODE
ROW 1	1	1	1	1	1	1	1	1	FF
ROW 2		1	1	1	1	1	1		7E
ROW 3			1	1	1	1			3C
ROW 4				1	1				18
ROW 5				1	1				18
ROW 6			1	1	1	1			3C
ROW 7		1	1	1	1	1	1		7E
ROW 8	1	1	1	1	1	1	1	1	FF

INPUT TO CHAR: "FF7E3C18183C7EFF"

Summary of Chapter 5

Chapter 5 has dealt entirely with computer graphics — the colorful patterns and designs you can create with your computer. Only two new statements have been introduced:

CALL SCREEN	Allows you to change the Run Mode screen to any color you choose.
CALL CHAR	Defines a character code for a character you create.

In addition to these two statements, you've experimented with the following techniques:

- Creating large color blocks on the screen
- Making patterns and designs from standard characters
- Animating your graphics
- Creating your own characters by turning dots "on" and "off"

The ability to create color graphics can add a lot of excitement to your computer programming. We hope that you've enjoyed this introduction to graphics and that you'll find even more creative ways to use your computer.

This chapter concludes our "introductory tour" through the TI BASIC language. You are now well launched into your programming career. For more advanced features you may want to consult the "BASIC Reference" section of the *User's Reference Guide*. Here are some sections we suggest:

- Additional editing techniques — see EDIT
- Automatic line numbering — see NUMBER
- BREAK
- TRACE
- Recording programs on the TI Disk Memory System or a cassette tape recorder — see SAVE and OLD

If you'd like to consult a programming book on an intermediate level, we can recommend an excellent one: Herbert Peckham's *Programming BASIC with the TI Home Computer* (New York: McGraw-Hill Book Company, 1979). You'll find a coupon for ordering this book on page 143.

Congratulations and best wishes for continued success in BASIC programming!

APPENDIX A
Musical Tone Frequencies

The tones produced by the computer are generated by the CALL SOUND statement. (See Chapter 1 for an explanation of the CALL SOUND statement.)

The *frequency* designated in the CALL SOUND statement determines the tone that is produced. The acceptable value range for frequencies is from 110 to 44,733 Hertz (cycles per second). Noninteger entries within this range are acceptable as inputs in the CALL SOUND statement, but they are rounded to nearest integers by the computer before execution.

The following table gives frequency values (rounded to integers) for four octaves in the tempered scale (one half-step between notes). While these values do not, of course, represent the entire range of tones — or even of musical tones — they can give you a basis for musical programs. (See Appendix D for a frequency-generating program.)

Frequency	*Note*	*Frequency*	*Note*
110	A	440	A (above middle C)
117	A#,B♭	466	A#,B♭
123	B	494	B
131	C (low C)	523	C (high C)
139	C#,D♭	554	C#,D♭
147	D	587	D
156	D#,E♭	622	D#,E♭
165	E	659	E
175	F	698	F
185	F#,G♭	740	F#,G♭
196	G	784	G
208	G#,A♭	831	G#,A♭
220	A (below middle C)	880	A (above high C)
220	A (below middle C)	880	A (above high C)
233	A#,B♭	932	A#,B♭
247	B	988	B
262	C (middle C)	1047	C
277	C#,D♭	1109	C#,D♭
294	D	1175	D
311	D#,E♭	1245	D#,E♭
330	E	1319	E
349	F	1397	F
370	F#,G♭	1480	F#,G♭
392	G	1568	G
415	G#,A♭	1661	G#,A♭
440	A (above middle C)	1760	A

Character Codes

All characters that print on the screen (letters, numbers, and symbols) are identified by numeric *character codes*. The standard characters are represented by character codes 32 through 127. These ninety-six codes are grouped into twelve *character sets* for color graphics purposes.

Standard Character Codes

Set #1		**Set #2**		**Set #3**		**Set #4**	
Code #	*Character*	*Code #*	*Character*	*Code #*	*Character*	*Code #*	*Character*
32	(space)	40	(	48	0	56	8
33	!	41	)	49	1	57	9
34	"	42	*	50	2	58	:
35	#	43	+	51	3	59	;
36	$	44	,	52	4	60	<
37	%	45	-	53	5	61	=
38	&	46	.	54	6	62	>
39	'	47	/	55	7	63	?

Set #5		**Set #6**		**Set #7**		**Set #8**	
Code #	*Character*	*Code #*	*Character*	*Code #*	*Character*	*Code #*	*Character*
64	@	72	H	80	P	88	X
65	A	73	I	81	Q	89	Y
66	B	74	J	82	R	90	Z
67	C	75	K	83	S	91	[
68	D	76	L	84	T	92	\
69	E	77	M	85	U	93	]
70	F	78	N	86	V	94	^
71	G	79	O	87	W	95	_

Set #9		**Set #10**		**Set #11**		**Set #12**	
Code	*Character*	*Code*	*Character*	*Code*	*Character*	*Code*	*Character*
96	`	104	H	112	P	120	X
97	A	105	I	113	Q	121	Y
98	B	106	J	114	R	122	Z
99	C	107	K	115	S	123	{
100	D	108	L	116	T	124	|
101	E	109	M	117	U	125	}
102	F	110	N	118	V	126	~
103	G	111	O	119	W	127	DEL

There are thirty-two additional character codes (128-159) available for use in defining special characters for graphics programs. (See Chapter 5 for a discussion of character definition.) Again, these codes are grouped into four sets for color graphics.

Special Character Codes

Set #13	**Set #14**	**Set #15**	**Set #16**
128	136	144	152
129	137	145	153
130	138	146	154
131	139	147	155
132	140	148	156
133	141	149	157
134	142	150	158
135	143	151	159

APPENDIX C

Color Codes

Sixteen colors are available for color graphics programs in TI BASIC. These colors are designated by numeric codes in the CALL COLOR and CALL SCREEN statements. (See Chapter 2 for a discussion of CALL COLOR and Chapter 5 for an explanation of CALL SCREEN.)

Color Codes

Color	*Code #*	*Color*	*Code #*
Transparent	1	Medium Red	9
Black	2	Light Red	10
Medium Green	3	Dark Yellow	11
Light Green	4	Light Yellow	12
Dark Blue	5	Dark Green	13
Light Blue	6	Magenta	14
Dark Red	7	Gray	15
Cyan	8	White	16

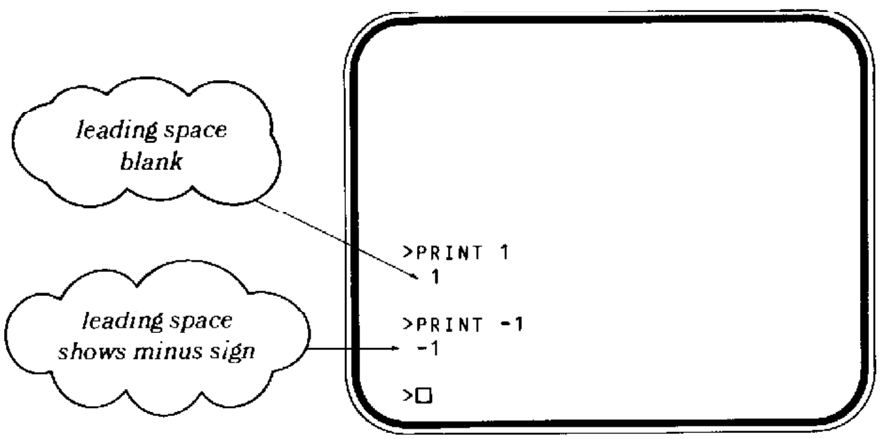
Mathematical Operations

If your computer is to be a useful tool, you'll need to know about some of its computational powers. This appendix first discusses the ways your computer handles and displays numbers and then shows you how to perform exponentiation (powers and roots of numbers). Next is a section on the order in which mathematical operations are performed. Finally, certain other mathematical functions are listed for you. You'll find that your computer can eliminate much of the drudgery of computation, leaving you with more time to explore the theory and fun of mathematics.

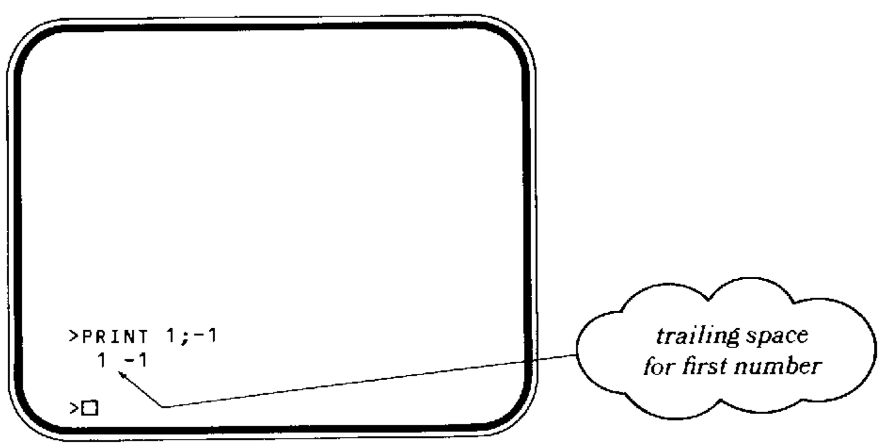
Decimal Notation

The Texas Instruments computer accepts and displays numbers, within certain limits, in the traditional decimal form.

In Chapter 3, we mentioned briefly that numbers are displayed with a *leading space* and a *trailing space*. The *leading space* is reserved for the *sign* (positive or negative) of the number. If the number is positive, this space will be blank. If the number is negative, this space will show a minus sign. Here's an example of both situations:



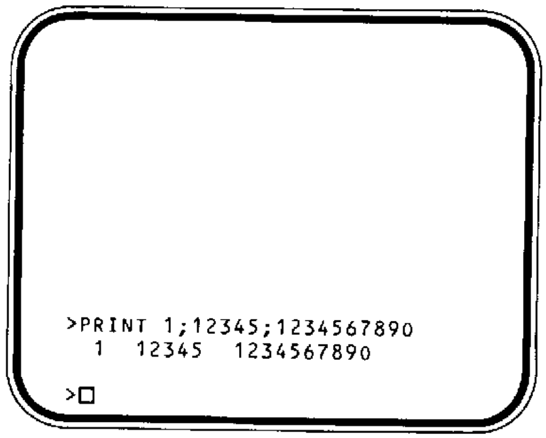
The *trailing space* is there to make sure that two numbers on the same line of the screen will always have at least one space between them, even if you use a semicolon as a PRINT separator. (The semicolon instructs the computer to leave no spaces between PRINT items.) Try this Immediate Mode example to see the effect of the trailing space:



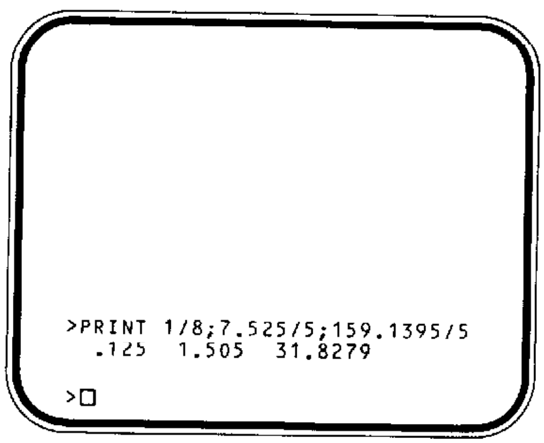
Without this trailing space the two numbers would appear like this:

1-1

The screen shows a maximum of ten digits for any number. If an integer (whole number) consists of ten digits or less, the computer shows the number *without* a decimal point to the right:



If the number is a decimal fraction with ten digits or less, the computer automatically places the decimal point in the correct position:



Notice the first example above, 1/8 = .125. If a number is less than +1 and greater than −1, so that the digit to the left of the decimal point would be zero, the zero is not displayed.

Most of the time, the numbers you see and work with will be shown in this normal display format. But what about numbers that consist of *more* than ten digits, such as

723,895,274,524
0.00000000014896

The computer can also handle numbers like these, but it must use a special display format to do so.

Mathematical Operations

Floating Point Form or Scientific Notation

To display numbers with more than ten digits, your computer uses a special kind of notation. You'll see several names in computer books referring to this type of notation: two of the more common names are *floating point form* and *scientific notation.* Here we'll refer to the special display format as scientific notation.

Before we discuss scientific notation, let's try a program to see how whole numbers (integers) look in this display format. Enter these lines:

```
NEW
10 LET A=10
20 FOR I=1 TO 12
30 PRINT A
40 LET A=A*10
50 NEXT I
60 END
```

Now clear the screen and run the program. You'll see these results:

```
10
100
1000
10000
100000
1000000
10000000
100000000
1000000000
1.E+10
1.E+11
1.E+12
```

Scientific notation starts here.

As soon as the value of A becomes an integer with more than ten digits, the computer switches over to the special display format. Here's what this format represents:

1.E+10 means 1×10^{10} or 10,000,000,000
1.E+11 means 1×10^{11} or 100,000,000,000
1.E+12 means 1×10^{12} or 1,000,000,000,000

Numbers that are printed in scientific notation will always have this form:

base number E exponent

The base number (*mantissa*) is always displayed with one digit (1 through 9) to the left of the decimal point. There can be a maximum of six digits in the mantissa (one to the left of the decimal point; up to five to the right of the decimal). "E" stands for "× (times) 10 raised to some power," and the *exponent* (power) is always displayed with a plus or minus sign (+ or −) followed by a one- or two-digit number (1 through 99).

Note: If you attempt to print a number with an exponent greater than 99 but less than the computer's limits, you'll see this format:

mantissa E+**
or
mantissa E−**

The two asterisks indicate that the number is within the valid computing range of the computer, but the exponent is too large to be displayed in the allotted space. (For a discussion of the computational ranges, see the "BASIC Reference" section of the *User's Reference Guide.*)

Here are several examples of integers that are displayed by the computer in scientific notation:

```
>PRINT 1234512345123
  1.23451E+12

>PRINT 45678900000000
  4.56789E+13

>PRINT 98765432100
  9.87654E+10

>□
```

Notice that the *sign* of the exponent tells us how to convert scientific notation back into standard decimal form. If the sign is a +, we move the decimal point to the right. If the sign is a −, we move the decimal point to the left. The *exponent* tells us how many places to move the decimal point:

1.11111E+10 means 11111100000

We have moved the decimal ten places to the right:

11111100000.

Integers with more than ten digits, then, are always displayed in scientific notation. Now let's see how the computer handles noninteger numbers (numbers with fractional parts). Consider the number 0.000000000000123. It will not fit into the ten-digit display, so the computer shows it in scientific notation. Try this:

```
>PRINT 0.000000000000123
  1.23E-13

>□
```

Tells how many places to move the decimal to the left.

The following program generates some very small noninteger numbers:

```
NEW
10 LET A=10
20 FOR I=1 TO 14
30 PRINT A
40 LET A=A/10
50 NEXT I
60 END
```

Clear the screen and run the program. The results are:

```
10
1
.1
.01
.001
.0001
.00001
.000001
.0000001
.00000001
.000000001
.0000000001
1.E-11
1.E-12
```

This program and the previous examples we've seen might lead us to think that nonintegers with more than ten digits are always displayed in scientific notation, just as integers are. This is not always true, however. *Noninteger numbers with more than ten digits are printed in scientific notation only if they can be presented more accurately in scientific notation than in the normal form.*

This point is very important. Consider an example that we've tried before:

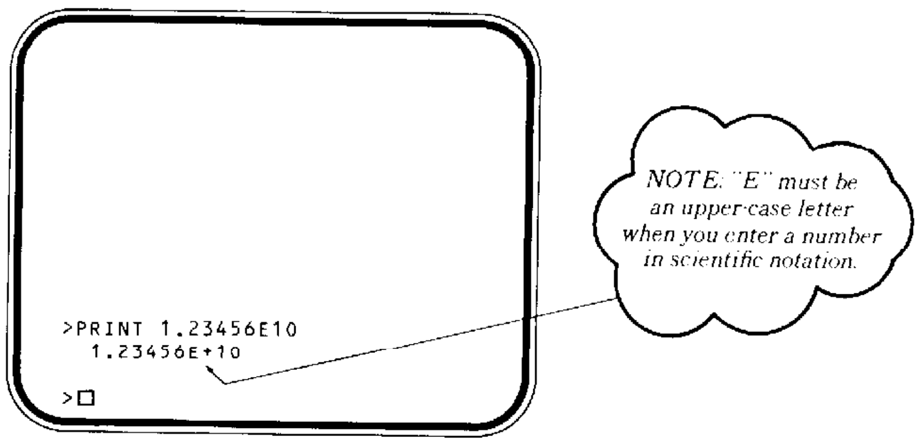
```
>PRINT 1/3
  .3333333333

>□
```

We know that .3333333333... is a repeating decimal that goes on infinitely. Why, then, does the display show the result in normal form? The answer is that .3333333333 is more accurate than 3.33333E−1; that is, more *significant digits* (digits that reflect the actual mathematical value of the number) can be shown in normal form than in scientific notation.

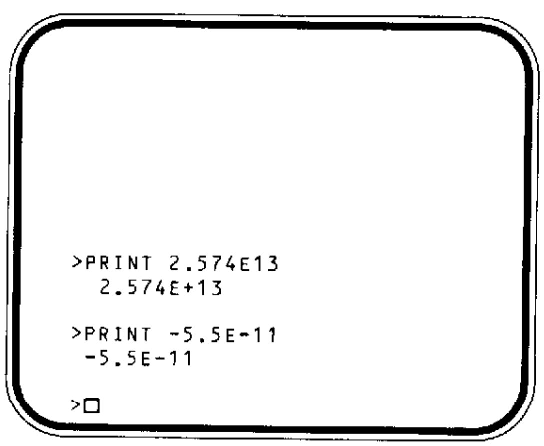
Scientific notation is just a "shorthand" method for writing long numbers, whether they are very large or very small quantities. It allows the computer to handle, in the most accurate form possible, numbers that otherwise could not be adequately displayed in the ten-digit normal form.

Entering Numbers in Scientific Notation

Up to this point, we've only entered numbers in the normal decimal form. It is also possible, however, to enter numbers in scientific notation. Try this example:



Notice that, unless you enter a minus sign before the mantissa and/or the exponent, these are assumed to be positive.



If you enter a number in scientific notation, but the computer can show it in normal form, it will do so. Try this:

```
>PRINT 5.555E3
 5555

>□
```

Whenever you are using extremely large or small numbers in a computation, entering the numbers in scientific notation can be very handy.

Exponentiation

In the previous section we talked about *exponents* and *powers* of 10. Now we need to discuss some of the "higher math" capabilities of your computer; specifically, *powers* and *roots.*

Powers

Quite often in mathematical calculations, we must raise some number to a *power*, such as

8^3 (or $8 \times 8 \times 8$)
25^2 (or 25×25)

To perform exponentiation (raising a number to a power) on the computer, we do this:

```
>PRINT 8^3
 512

>PRINT 25^2
 625

>□
```

The *exponentiation symbol* (^) tells the computer that the number that follows is a power.

Mathematical Operations

Let's say that we have this mathematical expression to evaluate:

$$y=x^3$$

We want to find all the values for y where x equals 1 through 10. So we enter this short program:

```
NEW
10 CALL CLEAR
20 FOR X=1 TO 10
30 Y=X^3
40 PRINT "Y=";Y
50 NEXT X
60 END
```

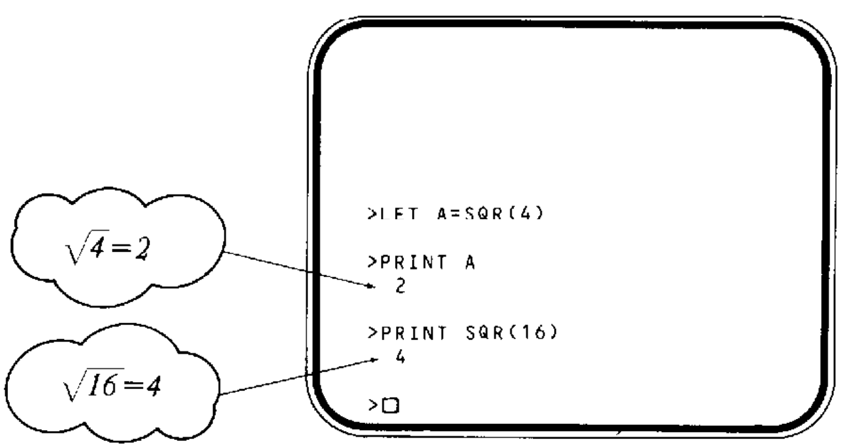
When we run the program, we'll see the following values for y:

```
Y= 1
Y= 8
Y= 27
Y= 64
Y= 125
Y= 216
Y= 343
Y= 512
Y= 729
Y= 1000
```

The computer completes the program for us very quickly! We have the values we need and can go on to other computations.

Roots

Finding a *root* of a number is another very common mathematical problem. The *square root* is one we've all heard of – and probably used – at some point in our educations. Since many, many calculations call for square roots, this function is built into the computer:



Mathematical Operations

The letters SQR stand for "square root of" and instruct the computer to find the square root of the number or expression contained within the parentheses.

Other roots must be computed by using a form of exponentiation. Computing a root of a number is the same function as raising the number to a power which is the reciprocal of the root; that is,

$\sqrt[3]{125}$ is the same as $125^{(1/3)}$

Try this example:

```
>PRINT 125^(1/3)
 5.

>□
```

Notice that we had to use parentheses around the exponent 1/3. The parentheses notify the computer that the whole expression makes up the exponent. (You'll see why this is necessary when we discuss "Order of Operations.")

Here's a program that helps you compute any root of any number (within the computer's limits and the bounds of mathematical rules, of course).

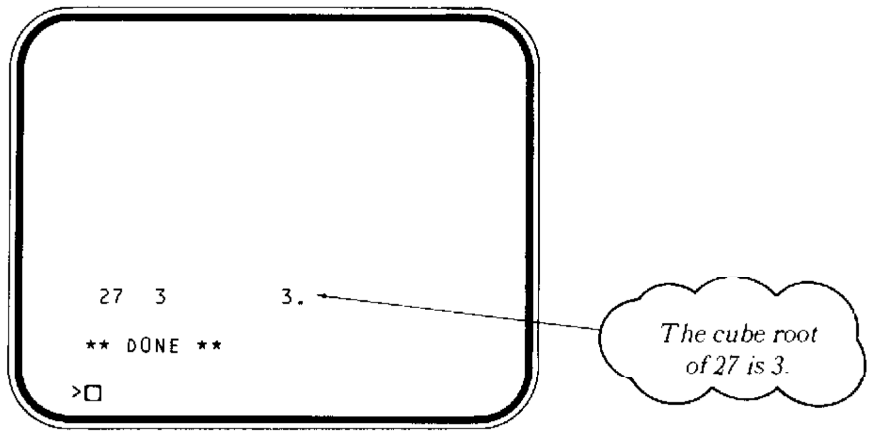
```
NEW
10 CALL CLEAR
20 INPUT "NUMBER?":N
30 INPUT "ROOT?":R
40 CALL CLEAR
50 PRINT N;R,N^(1/R)
60 END
```

comma

semicolon

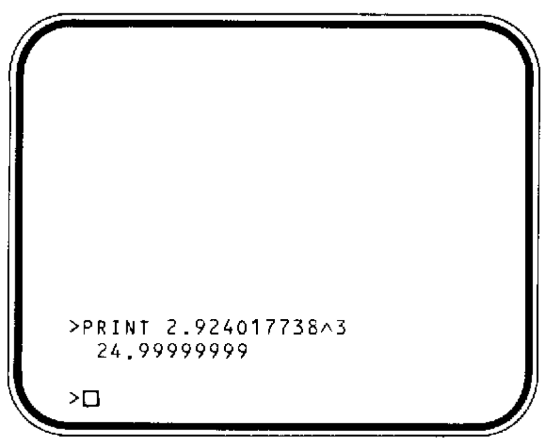
When you run the program, you'll first be asked to input the number for which you want to find the root. Let's enter *27* for our example.

Next you're asked for the root you want to find. Let's say we want the *cube root*, so we type *3* and press **ENTER.**



Run the program again, and this time enter *2401* for the number and *4* for the root. Did you get the answer 7?

Of course, not all numbers work out to results that are nice, neat integers. Try the program again, entering 25 for the number and 3 for the root. You'll get 2.924017738 as your answer. Now check the answer in the Immediate Mode, by raising 2.924017738 to the power of 3:



You don't quite get back to your original 25. That's because 2.924017738 is not the "exact" cube root of 25; it's an "approximate" root, rounded to ten digits so that it can be displayed.

All computing devices must "round off" calculated results at some point. Where a computer rounds a result depends on the computational and display limits of the machine. To make sure that the accuracy of the last displayed digit is within certain limits, most computers and many calculators actually perform computations internally with more digits than they can display. These extra or "guard" digits are retained in the computer's internal registers, but they can't be shown on the screen, due to space limitations.

We can, however, demonstrate the presence of these internal "computational" digits. Let's use the same problem we performed earlier:

```
>LET A=25^(1/3)

>PRINT A
  2.924017738

>PRINT A^3
  25.

>□
```

The "memory box" labeled A retains all the internal digits as well as the rounded result shown on the screen. Therefore, with the greater accuracy provided by the internal digits, we get back our original 25 when we raise A to the power of 3.

One special note of caution: Your computer will give you an error message if you try to raise a negative number of a fractional power; therefore, you cannot use the exponentiation routine to find roots of a negative number *without* taking other steps. See the Sign (SGN) and Absolute Value (ABS) functions in the "BASIC Reference" section of the *User's Reference Guide.*

Order of Operations

In Chapter 3 we discussed the order the computer follows to complete problems involving multiplication, division, addition, and subtraction. We also demonstrated that an expression within parentheses is evaluated before the rest of the problem is solved. The order of operations, then, was listed as:

(1) Complete everything inside parentheses.
(2) Complete multiplication and division.
(3) Complete addition and subtraction.

Now we need to add another level to this order. Exponentiation (raising a number to a power or finding a root of a number) is performed before any other mathematical operation. So our new order becomes:

(1) Complete parenthetical expressions.
(2) Complete exponentiation.
(3) Complete multiplication and division.
(4) Complete addition and subtraction.

Let's try some examples that help to demonstrate these concepts.

APPENDIX D

Mathematical Operations

First, we'll define some variable names for the quantities we'll be using in our calculations. Enter these lines:

```
LET A=5
LET B=2
LET C=10
LET D=4
```

Now we're ready for the calculations:

```
>PRINT B*C^B
  200

>PRINT A+B*C^B
  205

>PRINT ((A+B)*C)^B/D
  1225

>□
```

Here's the order the computer followed in each of these examples:

First problem	$10^2=100$ $2\times100-200$
Second problem	$10^2=100$ $2\times100=200$ $5+200=205$
Third problem	$5+2=7$ $7\times10=70$ $70^2=4900$ $4900\div4=1225$

Notice that this last problem utilized two sets of parentheses, one within the other. In this situation the computer evaluates the innermost set of parentheses first.

Mathematical Operations

As you saw when we discussed the roots of numbers, the exponent of a number can also be a numeric expression enclosed in parentheses. Let's try a few more examples, using the values already stored in the computer's memory.

```
>PRINT ((A+B)*(A+B))∧(B/D)
  7.

>PRINT B∧(D/B)+A*C
  54.

>□
```

The first problem essentially squared the number 7 and then took the square root of the result:

$(A+B)=5+2=7$
$(A+B)*(A+B)=7\times 7=49$
$B/D=2\div 4=.5$
$49^{.5}=\sqrt{49}=7$

The second problem is solved like this:

$D/B=4\div 2=2$
$B \wedge (D/B)=2^2=4$
$A*C=5\times 10=50$
$4+50=54$

The following program not only demonstrates the computational power of your computer, but also plays a scale for you!

The relationship between the frequencies of notes in the tempered scale can be algebraically expressed as

$y=xk^n$

where x = the frequency of the first note of the scale,
k = a constant, $\sqrt[12]{2}$,
n = the number of half-steps between note x and note y
y = the frequency of the next note you want to play

APPENDIX D
Mathematical Operations

There are twelve notes in the tempered scale, and between each note and the next is one half-step. The following program, starting with a frequency of 440 (A above middle C on a piano keyboard), calculates and plays each note in the scale:

```
20 X=440
30 K=2^(1/12)
40 CALL SOUND(200,X,2)
50 FOR N=1 TO 12
60 Y=X*K^N
70 CALL SOUND (200,Y,2)
80 NEXT N
90 END
```

Beginning frequency.

Calculates the constant, K.

Plays first note.

Calculates Y, the next note in the scale.

Plays note

Loops back to play next note.

Run the program and listen to the music!

Other Mathematical Functions

Several other mathematical functions, in addition to those we've already covered, are available in TI BASIC. We won't discuss these in detail, but we want to list some of them for you, because they can be a great help in performing mathematics with your computer.

Trigonometric Functions

These trigonometric functions are available:

`SIN( )` – Finds the *sine* of the number or numeric expression enclosed in parentheses.

A number or numeric expression goes here.

`COS( )` – Finds the *cosine* of the number or numeric expression enclosed in parentheses.

`TAN( )` – Finds the *tangent* of the number or numeric expression enclosed in parentheses.

`ATN( )` – Finds the *arctangent* of the number or numeric expression enclosed in parentheses.

Note: All trigonometric functions are performed by the computer in *radians*, rather than *degrees*. Therefore, if your data is measured in degrees, you'll need to convert the measurement to radians before using it with the function. (To convert an angle from degrees to radians, multiply by $\pi/180$. To convert from radians to degrees, multiply by $180/\pi$.)

Mathematical Operations

Logarithms

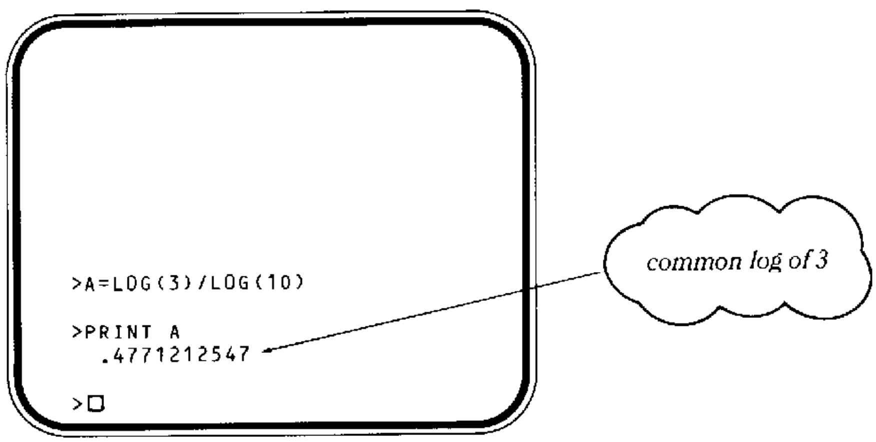
The computer calculates the *natural log* and *natural antilog* (based on e = 2.718281828) of a number:

A number or numeric expression goes here.

`LOG( )` Computes the natural logarithm of the number or numeric expression enclosed in parentheses.

`EXP( )` Computes the natural antilogarithm of the number or numeric expression enclosed in parentheses.

To convert the natural logarithm of a number to the *common log* of the number, simply divide the natural log by the natural log of 10. For example, if you want to find the common log of 3, you would use this procedure:



Absolute Value

Calculations often require the use of the absolute value of a number. This has the effect of making the number positive, regardless of its sign. Here's how to instruct the computer to find and utilize the absolute value of a number:

A number or numeric expression goes here.

`ABS( )` Finds the absolute value of the number or numeric expression in parentheses.

There are other mathematical functions available, and you'll find them listed and discussed under "Functions" in the "BASIC Reference" section of the *User's Reference Guide*. The functions we've illustrated here, however, should help you discover many ways to use your computer as a computational tool.

Index

6/81

Programming BASIC with the TI Home Computer

by Herbert D. Peckham

When you've completed *Beginner's BASIC*, you may want to explore further with the help of an intermediate level book. We recommend *Programming BASIC with the TI Home Computer* by Herbert D. Peckham (McGraw-Hill, 1979).

A well-known author and educator, Mr. Peckham has published numerous books on programming in BASIC. His new book takes you further into the full range and power of TI BASIC and your Texas Instruments computer. The easy-to-understand examples and relaxed style of the book can help you expand your programming skills and develop your own customized computer applications.

Use the coupon below to order *Programming BASIC with the TI Home Computer* from Texas Instruments Incorporated.

ORDERING INSTRUCTIONS

Shipping Inside U.S. Prepaid orders (check or money order) will be sent by postage-paid Third or Fourth Class postage. Allow 4 to 6 weeks for delivery. If you desire shipment other than Third or Fourth Class, additional postage should be included with your order along with specific directions as to method of shipment. Enter additional amount for shipping on order form.

Shipping Outside U.S. Orders prepaid in U.S. funds only will be accepted. Specify method of shipment and enter postage or shipping charge amount in proper blank on order form. This amount should be included in book payment.

Please send orders to: Texas Instruments Incorporated
P. O. Box 3640, M.S. 84M
Dallas, Texas 75285

To: Texas Instruments Incorporated
P. O. Box 3640, M.S., 84M
Dallas, Texas 75285

Please send me *Programming BASIC with the TI Home Computer.*

Name: ____________________

Address: ____________________

City: ____________ State: ________ Zip: ________

(LCB-4190) ____ copies @ $19.95 each .... ________

State and local sales tax (If applicable)* .... ________

Shipping method** ____________________

Additional postage enclosed** .... ________

Total payment amount enclosed .... ________

*State and local sales taxes required by every state except AK, DE, HI, MT, NH, OR.

**For mailing other than Third or Fourth Class.

Price effective October 1, 1982 subject to change without notice. BB-79

ADDENDUM
Accessing Speech from BASIC, Extended BASIC, and Terminal Emulator II

Important:

You must have the *Solid State Speech*™ Synthesizer peripheral plugged into the console to do any of the following:

- Run a BASIC program that accesses speech.
- Run an Extended BASIC program that accesses speech.
- Use the command **CTRL 1 (Speak)** while the Terminal Emulator II *Solid State Software*™ cartridge is installed.

If the Speech Synthesizer is not attached in any of these cases, your computer system will lock up. If this occurs, turn the console off and then turn off any peripheral devices. Wait several seconds. Turn on the peripheral devices and then turn on your console. This should restore your system to normal operation.

Note: Any data or program material that was not saved prior to the lock-up is lost.

1053227-1
(Use with 1039031-4)

Beginner's BASIC is a step-by-step guide that takes you from the "ground up" into an adventure — the adventure of communicating with a computer in a simple, yet powerful language. Quickly and easily you'll find yourself in control of a system that can calculate, make decisions, educate, and entertain. Even if this is the first time you've seen a computer, you'll be able to follow this easy-to-understand, hands-on approach.

It Starts at the Beginning. Getting started, do's and don'ts, all of the very first things you need to know. Simple but exciting first experiences with the computer make it fun from the start.

It Quickly Gives You Confidence. Although the approach is step-by-step, it's fast and bright. Carefully thought-out useful applications are built into the learning. In a short while you'll know all of the key elements of the BASIC computer language, as well as how to apply it in building powerful programs that carry out exactly what *you* want your computer to do. On completion of this book, you're ready to explore more advanced techniques explained in your *User's Reference Guide* or in the many other published materials available on BASIC programming.

A Learn-by-Doing Approach that shows you how to

- Command the computer in "Immediate Mode"
- Write simple programs and explore features that let you edit, calculate and make decisions
- Create graphic designs in 16 colors on the computer screen
- Build musical tones, patterns, and chords — in three voices with a range of more than four octaves each
- Develop more advanced programs that "put it all together" — computation, decision making, graphics and sound — to do the things that are most important to you!

An Important Part of Today's Literacy. Your Texas Instruments computer, along with this book, offers an excellent framework for learning about the world of computers. You'll know first hand what computers do — and what they can't do. You'll be familiar with the BASIC computer language — one of the most universally applied languages in the world. You'll be a computer "literate" — better able to discuss, understand, and decide on the role of computers in your life. In today's world "computer literacy" can give you an important edge as you encounter computer applications in education, entertainment, and business.

TEXAS INSTRUMENTS
INCORPORATED

Printed in U.S.A.

DALLAS, TEXAS

1039023-2